UTILIZING GRAPHICS PROCESSING UNITS FOR NETWORK ANOMALY DETECTION

THESIS

Jonathan D. Hersack, CIV

AFIT/GCO/ENG/12-24

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCO/ENG/12-24

UTILIZING GRAPHICS PROCESSING UNITS FOR NETWORK ANOMALY DETECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Insitute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Jonathan D. Hersack, B.S.C.S.E.

CIV

September 2012

AFIT/GCO/ENG/12-24

UTILIZING GRAPHICS PROCESSING UNITS FOR NETWORK ANOMALY DETECTION

Jonathan D. Hersack, B.S.C.S.E.
CIV

Approved:

Barry E. Mullins, PhD (Chairman) — 24 Aug 12 — Date

Rusty O. Baldwin, PhD (Member) — 24 Aug 12 — Date

Michael R. Grimaila, PhD, CISM, CISSP (Member) — 24 Aug 12 — Date

Abstract

Network intrusion detection signatures are often outdated as new attacks are developed more quickly than signatures. Machine learning anomaly-based network intrusion detection methods provide the ability to detect unknown attacks but require significantly more processing time than signature detection methods. The availability of Graphics Processing Units (GPUs) in many personal computers leads to a potential solution for a scalable, cost-effective network anomaly detection system. This research explores the benefits of using commonly-available graphics processing units (GPUs) to perform classification of network traffic using supervised machine learning algorithms.

Two full factorial experiments are conducted using a NVIDIA GeForce GTX 280 graphics card. The goal of the first experiment is to create a baseline for the relative performance of the CPU and GPU implementations of Artificial Neural Network (ANN) and Support Vector Machine (SVM) detection methods under varying loads. The goal of the second experiment is to determine the optimal ensemble configuration for classifying processed packet payloads using the GPU anomaly detector. The configurations include three base classifier configurations and two ensemble combination methods.

Experimental results show that the GPU implementation of the anomaly-based network intrusion detection system provides significant training and testing speedups over the CPU implementation for the Network Security Laboratory Knowledge Discovery and Data mining (NSL-KDD) dataset as well superior scaling across the load sizes evaluated. The GPU ANN achieves speedups of 29x over the CPU ANN. The GPU SVM detection method shows training speedups of 85x over the CPU. The GPU ensemble classification system provides accuracies of 99% when classifying network payload traffic, while achieving speedups of 2-15x over the CPU configurations. These results indicate the the GPU anomaly detection system is a viable solution for providing an additional layer in the defense of computer networks.

To my parents, who have always supported and encouraged me.

Acknowledgements

I would like to thank my advisor, Dr. Barry Mullins, for his encouragement throughout the research process. I would also like to thank Dr. Rusty Baldwin for his advice and feedback.

Jonathan D. Hersack

Table of Contents

List of Figures

List of Tables

Utilizing Graphics Processing Units for Network Anomaly Detection

1 Introduction

1.1 Motivation

Network Intrusion Detection Systems (NIDS) are a primary line of defense in securing computer networks against malicious traffic. Most NIDS today examine network traffic for known attacks using predefined signatures. These detectors sit at an ingress or egress point to the network and scan all passing traffic for matching patterns. Unfortunately, the signature-based detection scheme is limited to detecting known attack signatures. Newly developed attack vectors and zero-days are not detectable as the sensor has no signatures for them. As a result, the ability to detect novel attacks is negligible until new signatures are developed.

A second type of NIDS scheme known as anomaly-based detection creates a baseline for normal traffic and triggers on traffic that falls outside the normal operation of the network. This detection scheme has the potential to detect unknown intrusions or zero-day attacks that signature-based detectors are unable to detect. A current research trend in the field of anomaly-based intrusion detection uses machine learning algorithms. This research focuses on supervised machine learning algorithms which use labeled training datasets to create a generalization function that describes the relationship between the input features and the traffic classification. Two commonly used algorithms are neural networks, which create a generalized regression model for a data set, and support vector machines, which construct a maximum margin separator to split the data into classes. In practice, these detection techniques require a great amount of processing time for large datasets. However, with the advent of general purpose computing for GPU (GPGPU)

programming standards, it is possible to perform these computations on the GPU's highly parallelized architecture, significantly reducing the required processing time.

1.2 Overview and Goals

This research focuses on the implementation and evaluation of a graphics processing unit (GPU) accelerated network intrusion detection system (GNIDS) that uses the parallel nature of the GPU to perform network anomaly detection using supervised machine learning techniques. GNIDS is designed to support multiple machine learning classifiers in an ensemble setup using two different ensemble combination methods, a majority vote and a neural network classifier. The system is trained on a labeled training dataset and used to predict labels for future traffic. These predicted labels are written to a log for future review.

This research has three goals. The first goal is to determine the more accurate machine learning technique for classifying network traffic on the GPU between artificial neural networks (ANNs) and support vector machines (SVMs). To accomplish this goal, GNIDS is implemented using two CUDA machine learning libraries, GTSVM and LIBCUDANN [CSK11][Don11]. A GeForce GTX 280 is used to perform baseline comparisons of the machine learning techniques on the Network Security Laboratory Knowledge Discovery and Data mining (NSL-KDD) IDS evaluation dataset [TBL09b]. The second goal is to evaluate the performance differences between executing GNIDS's detection algorithm on the CPU and GPU. To accomplish this goal, GNIDS is also implemented using the LIBSVM CPU support vector machine library and the CPU ANN functionality of LIBCUDANN [ChL11]. The relative execution times are collected and compared for the NSL-KDD dataset and a dataset preprocessed using 2_ν-gram byte frequency analysis. The third goal of this research is to determine which ensemble configuration provides the highest accuracy when classifying payload data. To accomplish this goal, the ensemble functionality of GNIDS is used to classify the 2_ν-gram frequency

analysis dataset using the majority vote and final ANN classifier ensemble combination methods with three different configurations of base classifiers for each machine learning detection method. This research is unique in that it analyzes the performance of supervised machine learning intrusion detection techniques on the GPU. It also explores the feasibility of using supervised machine learning with the 2_v-gram payload analysis technique using ANN ensembles and SVM ensembles on the GPU.

The hypothesis of the first research goal is that support vector machines will be more accurate at classifying anomalous network traffic using the GPU than the artificial neural networks. The second goal hypothesizes that the GPU implementation for the intrusion detection system will be significantly faster than the CPU implementation. For goal three, the hypothesis is that adding more base classifiers to the ensemble and using a final classifier combination method instead of a simple majority vote will increase the accuracy of the ensemble when classifying network payload data processed using the 2_v-gram technique from McPAD [PAF09].

1.3 Thesis Layout

This chapter presents the topic, explores the motivation, and summarizes the goals of this research. Chapter 2 provides background information on network intrusion detection systems (NIDS), machine learning, and general purpose computing on graphics processing units (GPGPU) using NVIDIA graphics hardware. Chapter 3 presents the methodology used to evaluate the performance of GNIDS. The results of the experiments are presented and analyzed in Chapter 4. Lastly, Chapter 5 provides a summary of the conclusions drawn and a discussion of areas for future work. The data collected in Experiments 1 and 2 is included in Appendices B and D, respectively.

2 Background and Related Work

This chapter provides the background and related work for network intrusion detection using machine learning via general purpose computing on graphics processing units (GPGPU). Sections 2.1.1-2.1.3 give a brief overview of the history of intrusion detection and the various types of intrusion detection systems (IDS). Section 2.1.4 provides an overview of commonly used intrusion detection techniques. Section 2.2 presents an overview of machine learning. A description of the specific machine learning techniques used in this research is provided in Section 2.3. Section 2.4 discusses related research in the area of machine learning intrusion detection systems. Section 2.5 presents the basics of GPGPU and the CUDA programming model. Section 2.6 discusses related work with GPGPU machine learning and intrusion detection. Lastly, Section 2.7 describes the specific machine learning implementations used in this research.

2.1 Intrusion Detection

2.1.1 Development of the Intrusion Detection System. The concept of intrusion detection (ID) began in the 1980's with James Anderson's paper on computer security and threat modeling. In his paper, Anderson noted that unauthorized accesses could be detected through the use of audit files [And80]. In 1987, Dorothy Denning wrote a paper that outlined a methodology for intrusion detection using patterns of abnormal system usage. This paper is often credited with sparking the imagination of researchers in the intrusion detection field, leading to the development of several intrusion detection techniques. The premise for intrusion detection provided by Denning is that intrusions are indicated by some abnormal use of the target system and should be detectable [Den87].

Intrusion detection techniques are distinguishable by the source of data they analyze and the method with which they analyze it [Kum07]. The first distinction separates intrusion detection systems into host-based, distributed, and network-based solutions.

Host-based detectors focus on system event logs and system calls. Distributed detectors collect audit information from several systems and their interconnecting network [NiJ03]. Network-based detectors focus on analyzing the contents of network packets directly for matching traffic [Kum07]. These intrusion detection systems are split by their detection method into two types, misuse detection systems and anomaly detection systems.

2.1.2 Misuse Detection. Misuse detection, often called signature-based detection or rule-based detection, operates by comparing network traffic to known signatures and patterns [Kum07]. Examples of common signatures are specific system commands used in an attack, specific request strings used by malware, or specific status codes or requests found in protocol headers and responses [ScM07]. The primary drawback of misuse detection is the dependency on known attack and misuse patterns. Before the IDS can detect new attacks, its signatures must be updated for the patterns of the new attacks.

Signature-based detectors use a variety of techniques. A commonly used technique is the Aho-Corasick algorithm [Kum07]. This algorithm is popular because it allows for string matching in linear time relative to the input. It operates by constructing a finite automaton from the signatures and allows them to be searched in multiple stages. A second popular technique is regular expression signatures. Regular expressions are popular due their added flexibility as compared to fixed string signatures. Since this research is primarily focused on anomaly-based detection, signature detection techniques are not discussed in detail.

2.1.3 Anomaly Detection. Anomaly detection consists of two stages, a training stage, in which the baseline of normal behavior is established, and an analysis stage, in which the traffic is compared to the baseline or profile [PaP07]. If traffic deviates significantly from the system's normal profile, it is classified as an attack. Since anomaly detectors are not based on known attacks but on the comparison of normal and anomalous

behavior, they have the potential to detect previously unknown attacks [Kum07]. Like misuse detectors, anomaly detectors have well-known drawbacks. First, it is possible to create a baseline that contains unidentified attacks, resulting in their classification as normal behavior [NiJ03]. Also, it can be difficult to draw a clear distinction between normal and anomalous behavior, resulting in a larger number of false positive alerts when compared to a misuse detector [PaP07]. Regardless of these drawbacks, anomaly-based detectors have the potential to increase the overall security of computer networks as future attacks are developed more quickly than new signatures. This is the primary motivation for this research which explores the use of GPGPU anomaly-based detectors as a method of adding additional layers of computer network defense.

2.1.4 Common IDS Techniques. There are numerous techniques used for anomaly-based intrusion detection. This section provides an overview of some commonly used techniques. The first technique is statistical anomaly detection. This technique examines the statistical properties of the traffic such as the mean, variance, and limits to classify traffic data. By collecting metrics such as the number of distinct IP addresses, CPU load, connection rate, and login attempts, the detector builds a statistical model for the normal behavior of the system [GDM09]. As the system runs, the detector collects a current traffic profile and compares its stochastic properties to the original baseline profile, generating alerts when significant differences are detected. Haystack, IDES, and SPADE are some examples of IDS that use statistical detection methods [PaP07].

Data mining is a second technique commonly used for anomaly detection. Data mining algorithms, such as fuzzy logic, are used to determine patterns from the training data and create baselines for the system. The Fuzzy Intrusion Recognition Engine (FIRE) uses fuzzy logic to create a set of fuzzy rules that define attacks against the network based on observed patterns in the input features [PaP07]. Genetic algorithms are another data mining technique that is commonly used. These algorithms are based on probabilistic

rules and evolve to converge on a general solution from several directions [Kum07][RuN09]. They are often used to derive classification rules and to optimize parameters used in other detection algorithms [GDM09].

Clustering is another data mining method used for anomaly detection. Clustering techniques group traffic data into clusters based on a distance metric [GDM09]. Some clustering based techniques train the system using unlabeled data, meaning that the system is not told which traffic is normal or anomalous. The assumption is made that the proportion of anomalous traffic is smaller. Traffic belonging to the smaller clusters is considered anomalous by the system. Other methods perform clustering on normal data only and create a profile. New traffic is then evaluated to see if it fits into the normal cluster profile [PaP07].

Many anomaly detection solutions use machine learning algorithms. Commonly used techniques include artificial neural networks, Bayesian networks, Markov models, and support vector machines. Support vector machines and artificial neural networks are the primary focus of this research and are discussed in Sections 2.3.1 and 2.3.3.

2.2 Machine Learning

The study of machine learning is the attempt to create a machine agent that can "improve its performance on future tasks after making observations about the world" [RuN09]. In most machine learning approaches, agents learn by analyzing a set of training data and generalizing the input-output relationships of the training set. This generalization is used to predict outcomes based on future input or classify examples with a predicted label.

2.2.1 Learning Types. There are three main categories of learning techniques, distinguished by their feedback approaches [RuN09]. The first of these types is unsupervised learning. In this approach, the agent is presented with unlabeled data. The

agent must make its own determinations as to the relationships of the data presented to it. The second type of learning is reinforcement learning. In this approach, the agent receives positive or negative feedback when it correctly, or incorrectly, reaches a conclusion. The agent determines which step in the process resulted in the punishment and attempts to correct its reasoning [RuN09]. The last formal category is supervised learning. Supervised learning is similar to reinforcement learning in that the agent is given feedback on input-output pairs. However, in supervised learning, the agent is presented with a pre-labeled training set that it uses to form its generalizations [RuN09].

2.2.2 Supervised Learning. The machine learning algorithms used in this research utilize supervised learning as their feedback method. Supervised learning can be described mathematically as:

> Given a training set of N example input-output pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, where each y_j is generated by an unknown function $y = f(x)$, discover a function h that approximates the true function $f(x)$ [RuN09].

The approximating function h is called the hypothesis. The goal of a well-generalizing hypothesis is to successfully predict the output for future inputs that were not included in the training set without overfitting. Overfitting occurs when a hypothesis function fits the training data but is not useful for predicting the general case from novel data.

2.2.3 Cross-Validation. K-fold cross-validation is a technique commonly used to evaluate the performance of a machine learning hypothesis on predicting future values from novel data. This technique breaks the training data into k subsets or folds. The learning algorithm is trained on each of the subsets until only one subset remains, $\frac{1}{k}th$ of the data. This last set is used as a validation set to determine the error rate of the learning algorithm's hypothesis. The process is repeated k times with each of the k folds serving as

a test set for a round. The average error rate for the folds is used to evaluate how well a hypothesis predicted the output for the entire dataset [RuN09].

2.3 Selected Machine Learning Techniques

2.3.1 Support Vector Machines. Support Vector Machines (SVMs) are a machine learning technique that is a primary focus area for this research. SVMs operate by constructing a ”maximum margin separator”, a generalization function that separates example points into distinct regions as illustrated in Figure 2.1 [Fle09][RuN09]. In this figure, the data belongs to one of two classes, a black dot or white dot. The solid black line is the support vector that separates the data points into distinct groups and the distances from it to the dotted lines are the margins (d_1 and d_2). The goal of creating a support vector machine is to determine the support vector that separates the classes with the greatest margin between the groups.

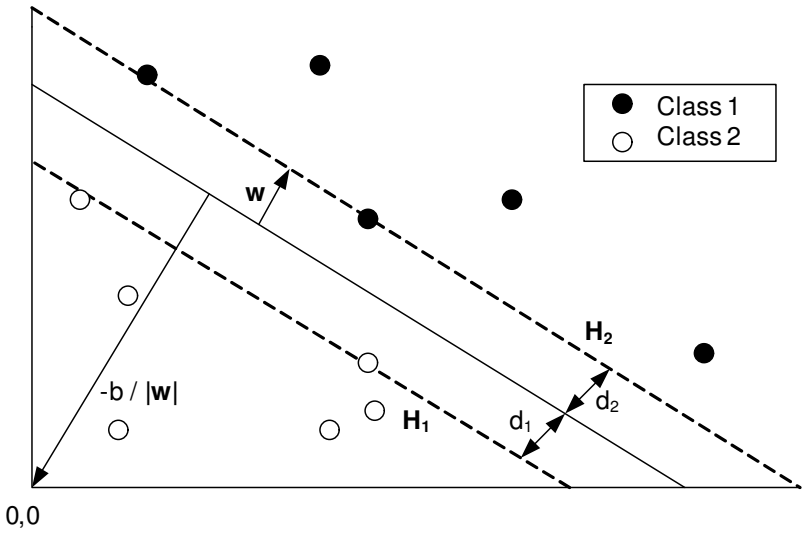

Figure 2.1: Support Vector Example

SVMs have a few key properties that provide advantages over similar types of machine learning. First, SVMs have the ability to map their data into a higher-dimensional

space, allowing non-linearly separable data to be separated in the higher dimensions. This technique is known as the kernel trick and is a result of using a non-linear function as the kernel function for the SVM. The second advantageous property of support vector machines is the way the training state is stored. SVMs are non-parametric with respect to training; all of the training samples are retained in the generalization model. However, support vector machines have the advantage over other non-parametric models in that they optimize on the more important training examples, allowing the number of retained support vectors, or margin separators, to be reduced. This results in a combination of non-parametric and parametric models, allowing SVMs to be resistant to overfitting [RuN09].

2.3.2 Theoretical Basis for SVM. As mentioned in the previous section, SVMs use maximal margin separators, also called hyperplanes, to separate the training data into classes. These separators can be described by $\mathbf{x_i} \cdot \mathbf{w} + b = 0$, where $\mathbf{w}$ is the normal to the hyperplane, $\mathbf{x_i}$ is a given training instance, and b is the perpendicular distance from the hyperplane to the origin as shown in Figure 2.1. Constructing SVMs comes down to selecting parameters w and b so that the training data is separated and described by the following equations [Fle09]:

$$\mathbf{x_i} \cdot \mathbf{w} + b \geq +1, \qquad y_i = +1 \tag{2.1}$$

$$\mathbf{x_i} \cdot \mathbf{w} + b \leq -1, \qquad y_i = -1 \tag{2.2}$$

and the combined form:

$$y_i(\mathbf{x_i} \cdot \mathbf{w} + b) - 1 \geq 0 \forall_i \tag{2.3}$$

where y_i is the class label for a given training instance. (2.1) is the boundary for the positive class and (2.2) is for the negative class. SVM implementations use quadratic programming optimization to search the space of w and b for a solution that correctly

separates the examples with the maximum margin. A full derivation of the equations is beyond the scope of this section and can be found in Fletcher's SVM tutorial [Fle09].

After the optimal separators have been computed, the equation for the separator (support vector) is given by [RuN09]:

$$h(x) = sign\left(\sum_j a_j y_j(\mathbf{x} \cdot \mathbf{x}_j) - b\right) \tag{2.4}$$

where *sign*() returns the sign of the classification. A key property of the final equation is that the data is represented by the dot products of pairs of points. This allows the kernel trick mentioned in Section 2.3.1 to be used to evaluate the dot products in a corresponding non-linear feature space without evaluating the full features of each data point [RuN09]. Additionally, a regularization, or cost, parameter is often added to the above equations, allowing a penalty to be assigned to misclassified points in the event that the data is not fully separable [Fle09]. This research uses a regularization parameter and a non-linear kernel. The non-linear kernel function used in this research is the Gaussian radial basis function. It is given by $e^{-\gamma * |\mathbf{x}-\mathbf{x_j}|^2}$ which replaces the dot product $(\mathbf{x} \cdot \mathbf{x}_j)$ in the linear kernel given in (2.4) [ChL11]. The Gaussian kernel has one parameter, gamma (γ), which determines the flexibility of the final margin separator [HuW10].

2.3.3 Artificial Neural Networks. The second machine learning techniques used in this research is artificial neural networks. Artificial Neural Networks (ANNs) are a type of machine learning that utilize a collection of nodes, called neurons, to predict a result based on a set of inputs. A neuron in an ANN is made up of an input function, an activation function, and an output. These neurons are linked together with various weights assigned to the links. Figure 2.2 illustrates a simple neuron [RuN09].

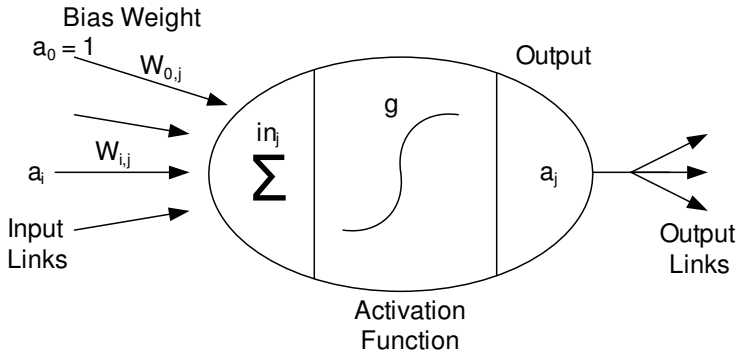

Figure 2.2: An Artificial Neuron

Artificial neural networks contain numerous neurons connected with weighted links. Each neuron computes a weighted sum of its input links using

$$in_j = \sum_{i=0}^{n} w_{ij} a_i \tag{2.5}$$

and applies an activation function to derive its output as follows.

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{ij} a_i\right) \tag{2.6}$$

The activation function is usually a hard threshold or a logistic function. Using non-linear functions allows the network to describe non-linear relationships between the inputs and outputs [RuN09]. A commonly used activation function is the sigmoid function. This function is given by [Nis12a]:

$$y = \frac{1}{1 + e^{-2x}} \tag{2.7}$$

A second commonly used function is the symmetric sigmoid or hyperbolic tangent function given by [Nis12a]:

$$y = tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.8}$$

This research uses both the sigmoid function and the hyperbolic tangent function for the activation functions of the neurons.

2.3.4 Feed-Forward Neural Network. This research uses a type of neural network known as a feed-forward neural network. Feed-forward neural networks consist of layers of neurons that feed their outputs forward as the inputs to the next layer of neurons. The neural networks used in this research consist of three layers, an input layer, a layer of hidden neurons, and an output layer as illustrated in Figure 2.3 [RuN09].

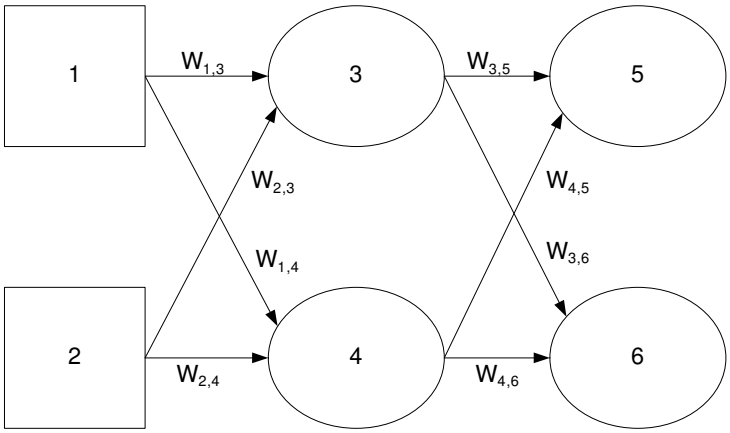

Figure 2.3: Example Multilayer ANN

The input layer, nodes 1 and 2, represents the inputs from the dataset to the network. This layer is connected by weighted links to the hidden neuron layer, neurons 3 and 4. Each neuron in the hidden layer performs the input aggregation and logistic activation described in (2.6). The outputs from the hidden layer neurons then feed forward as inputs to the output layer neurons, neurons 5 and 6. The output layer also computes an activation function to determine the outputs of each output neuron.

For inputs $x = (x_1, x_2)$, the output of neuron 5 in the example network depicted in Figure 2.3 is given by expanding (2.6) into

$$a_5 = g(w_{0,5}+w_{3,5}a_3+w_{4,5}a_4) = g(w_{0,5}+w_{3,5}g(w_{0,3}+w_{1,3}x_1+w_{2,3}x_2)+w_{4,5}g(w_{0,4}+w_{1,4}x_1+w_{2,4}x_2)) \tag{2.9}$$

where $g(x)$ is the activation function of the given neuron and the $w_{0,x}$ weights are the bias inputs with a value of 1 [RuN09]. This research uses the sigmoid function (2.7) as the activation function for the hidden layer neurons and the hyperbolic tangent function (2.8) as the activation function for the output layer neurons.

2.3.5 Training Feed-Forward Neural Networks. Feed-forward neural networks contain all of their hypothesis information in the weights assigned to the individual links between the neurons. In order to generalize a learning problem, the weights of the network must be selected that best describe the relationships between the input features and the output label. This research uses a training technique called back-propagation. Back-propagation consists of two-phases. The first phase is the computation of the output error and the second phase is the updating of the weights in the network to minimize that error for the given training set. The incremental version of the back-propagation algorithm is simplified as follows [RuN09]:

1. For the given training example, calculate the activation function outputs feeding forward to the last layer.

2. Back-Propagate the deltas to the input layer.

    a. For each output neuron j, calculate the delta given by $\Delta_{[j]} = g'(in_j) \times (y_j - a_j)$, where the vector $\mathbf{y}$ is the desired output.

    b. For each non-output layer

        i. For each neuron i in layer l, calculate the delta given by $\Delta_{[i]} = g'(in_i) \sum_j w_{i,j} \Delta_{[j]}$.

    c. Update the weights $w_{i,j}$ such that $w_{i,j} \leftarrow w_{i,j} + a \times a_i \times \Delta_{[j]}$.

3. Steps 1 and 2 are repeated until a stopping condition (such as target mean square error for the predicted outputs) is met or the target number of epochs (iterations) is reached.

This research utilizes a slightly different technique called batch back-propagation. The primary difference is incremental training updates the weights after each training example for every epoch, or repetition of the algorithm, while batch training computes the deltas for all of the examples before updating any of the weights, resulting in only one update per epoch. This results in slower training for some problems, but can be more accurate due to the more accurate calculation of the mean square error [Nis12b].

2.3.6 Ensembles. Machine learning techniques such as artificial neural networks and support vector machines may also be used in combination. Ensemble learning is a technique that combines multiple learning algorithm hypotheses. The principle behind ensembles is to reduce the overall chance of error by combining all of the independent hypotheses into one. Some techniques, such as boosting, involve adjusting the training data based on intermediate predictions, while others utilize a combination method such as a majority vote, sum, or product of the outputs.

Several common techniques used to combine classifiers are categorized as winner-take-all approaches. These include majority voting and weighted majority vote. Consider a system that has five hypotheses. In order for a misclassification to occur, three or more of its hypotheses have to agree on the incorrect answer [RuN09]. This is an example of a majority vote system. A weighted majority vote is similar to majority vote but the base classifiers each receive a weight to influence their input in the final voting. Additional techniques include naive-Bayes combination, sum averages, and neural network combination [CYT06].

This research focuses on the majority voting and neural network methods for combining classifier outputs. The neural network combination method takes the outputs of

the base classifiers as inputs and trains a classifier hypothesis function to best combine the outputs into a final prediction. This allows the weights of the base classifiers to be automatically adjusted unlike the weighted majority vote and weighted sum techniques which use a fixed set of weights for the base classifiers [CYT06].

2.4 Related Machine Learning IDS Work

Extensive work covering the use of various types and combinations of ANNs and SVMs in intrusion detection has been published. Silva et al. utilize hamming nets, a type of neural network, to classify network traffic payloads using signatures obtained from Snort, resulting in a 70% classification of illegitimate traffic [DFD04]. Zhang et al. present a framework which employs a statistical modeling technique to create profiles for network traffic and compare the performance of five types of neural networks on classifying traffic according to the generated profiles [ZLM01]. Golovko et al. use a combination of neural networks and principle component analysis techniques. These neural networks are used to perform feature reduction and classification in both a stand-alone configuration and an ensemble setup, observing accuracies of 93% [GVK07].

Mukkamala and Sung compare the relative performance of support vector machines and several types of artificial neural networks on the classification of preprocessed network traffic, achieving accuracies of 99% for SVMs and 97% for ANNs [MuS03]. John Mill conducts a comparison of several SVM techniques and their performance when classifying network traffic and presents a technique called ArraySVM which is similar to an ensemble of SVM. The SVM with the greatest margin, or confidence, when classifying a novel point is used to make the overall prediction, reaching accuracies of 91% for his evaluation set [MiI04]. Safaa Zaman takes an ensemble-like distributed approach to classifying network traffic. Multiple classifiers (SVM and ANN) are used with different input features targeted to specific layers of the Internet Protocol stack to detect specific types of traffic and attacks, achieving accuracies of 99% for his evaluation data [ZaK09].

These research efforts show that ANN and SVM techniques can be successfully applied to the classification of network traffic.

2.5 Graphics Processing Units

This section discusses the other focus area of this research, general purpose computing on graphics processing units (GPGPU). Section 2.5.1 provides an overview of modern graphics hardware. Section 2.5.2 introduces the NVIDIA GPGPU standard, compute unified device architecture (CUDA). The CUDA model is discussed in Section 2.5.3. Section 2.5.4 explains the CUDA compilation process and Section 2.5.5 covers the primary differences between generations of CUDA hardware.

2.5.1 Graphical Processing Units. Modern graphics hardware consists of one or more graphical processing units (GPU), onboard memory, and a PCI-express interface. With all of these dedicated components, graphics cards are effectively their own complete computing contexts. The key difference between central processing units (CPU) and GPUs is the nature of the computations that they handle. CPUs, like the Intel Core i7 series, are optimized for sequential operations, branch prediction, and quick cache operations, while GPUs are optimized for handling large amounts of streaming data and similar operations in parallel. The parallelism is possible because graphics processors have more transistors devoted to ”data processing rather than data caching and flow control” than traditional CPUs as illustrated in Figure 2.4 [NVI11b].

GPUs require this level of parallelism to accomplish their primary task, graphics rendering, which is a process with a higher arithmetic intensity than standard computing tasks [NVI11b]. Arithmetic intensity is the ratio of arithmetic operations to memory operations. Graphics rendering falls into the category of high arithmetic intensity because large numbers of repetitive mathematical operations are required to transform textures and matrix representations of objects into forms that are viewable on a two-dimensional

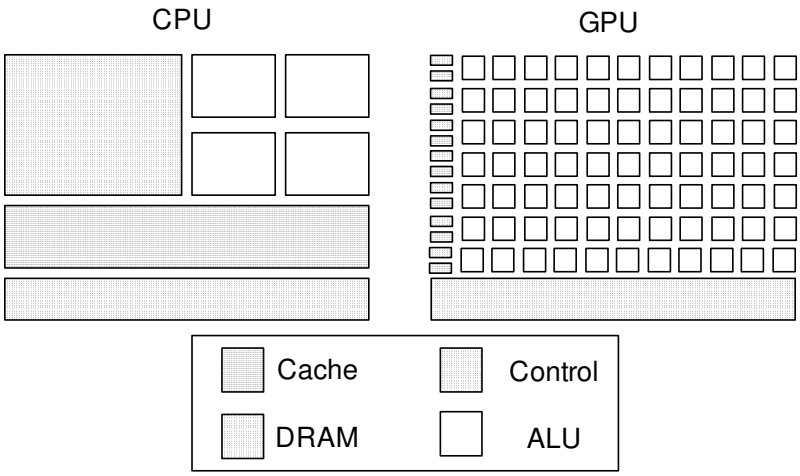

Figure 2.4: CPU Control versus GPU Control

display. These tasks are approached with data parallel processing which maps data elements to parallel threads which execute the same program for each element, hiding the memory access latency with calculations across multiple threads rather than with a cache as on a CPU [NVI11b].

2.5.2 General Purpose Computing on GPU. In 2006, NVIDIA, a leading graphics card manufacturer, released a new architecture designed to leverage the parallel nature of the GPU for general purpose programming. This architecture is referred to as compute unified device architecture (CUDA). CUDA GPUs implement an architecture type described by NVIDIA as a single instruction, multiple thread (SIMT) architecture. SIMT is similar to single instruction, multiple data (SIMD) in that it controls multiple processing elements with a single instruction. However, while the SIMD vector exposes the SIMD width to the software, the CUDA platform instructions specify the execution and branching behavior of a single thread [NVI11b]. This type of architecture allows CUDA programmers to focus on solving the programming problem at the thread-level using parallel code across scalar independent threads or coordinated threads where these threads

may operate on different data [NVI11b]. These segments of parallel code are called CUDA kernels.

2.5.3 CUDA Model. The CUDA hardware architecture consists of a scalable array of streaming multiprocessors (SMs). The multiprocessors are responsible for creating, managing, scheduling, and executing groups of threads. CUDA organizes its threads into three basic layers: the thread, the thread block, and the grid of thread blocks. The CUDA thread hierarchy is depicted in Figure 2.5.

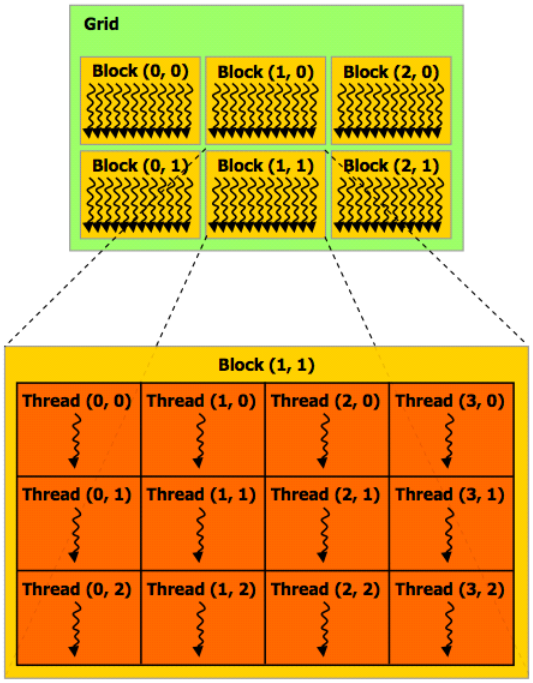

Figure 2.5: CUDA Thread Hierarchy

The CUDA thread is the lowest level of the thread hierarchy. These are lightweight threads with little overhead, unlike general purpose threads on a CPU which have higher context switching penalties [NVI12a]. Threads are organized into groups of 32 known as warps. All threads in a warp follow the same execution path, executing the same instruction in parallel with their respective data. For example, consider the pseudo-code

kernel snippet in Figure 2.6. If only twenty threads in a warp meet the condition for the if block, twelve threads will be paused awaiting the execution of the if case by the first twenty threads. Next, the twenty threads will pause while the remaining twelve execute the else case. Finally, when all of the separate paths converge, all the threads can execute in parallel once again. It should be noted that this branch divergence cost occurs only within a warp; separate warps always execute independently. Maximum efficiency is reached when all threads in a warp agree on a common execution path [NVI11b].

```
If(thread level condition)
{
        Do work1;
}
Else
{
        Do work2;
}
Do work3;
```

Figure 2.6: Branching Example

As shown in Figure 2.5, CUDA threads are organized into equally-sized blocks. These thread blocks must be able to be executed in arbitrary order, in parallel, or in series. As a result of this independent execution requirement, the CUDA kernels can be easily distributed across multiple multiprocessors and block level branching does not incur the cost of lower-level branching. Figure 2.7 illustrates the way that these blocks can be distributed across GPUs with differing numbers of cores [NVI11b]. Each GPU core is assigned a group of blocks to execute. GPUs with fewer cores will execute kernels more slowly as the thread blocks must wait for an available core.

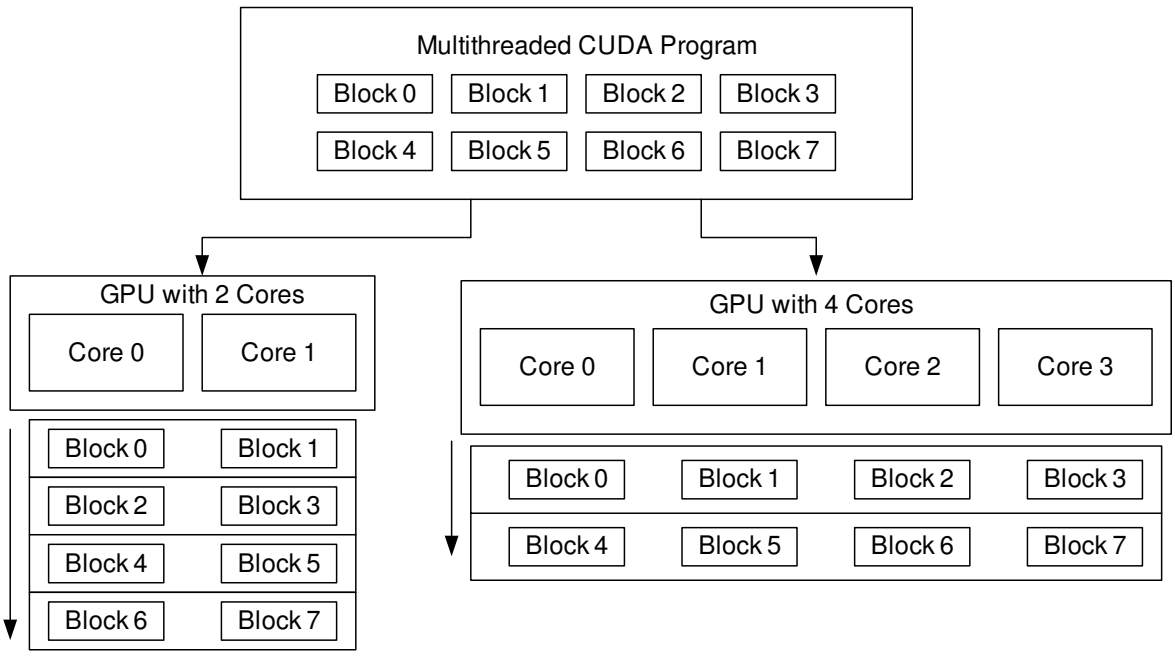

Figure 2.7: CUDA Thread Block Scaling

Thread blocks are assigned to a grid, the highest level in the thread hierarchy. When a CUDA kernel is executed, it is assigned a grid of thread blocks which are assigned to the available SMs and partitioned for execution into warps [NVI11b]. CUDA programs execute in a heterogeneous environment; a portion executes serially on the CPU, while the parallel kernels are assigned grids of thread blocks and executed on the CUDA device as shown in Figure 2.8.

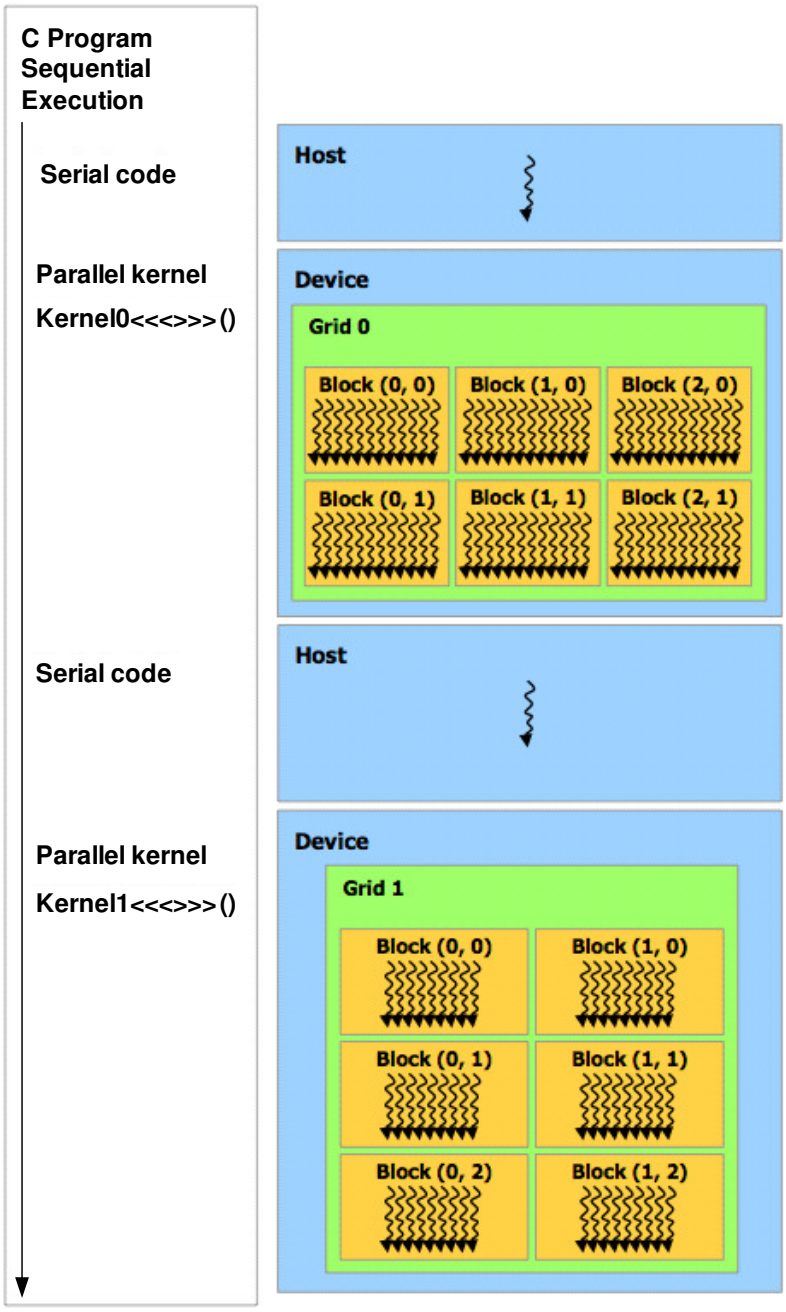

Figure 2.8: CUDA Kernel Execution

Like the threading model, CUDA memory is also broken into layers with each layer accessible by certain layers in the thread hierarchy as shown in Figure 2.9. Each thread has access to its own registers and its own private local memory. Blocks have access to

shared memory accessible by all the threads in the block. Global memory is accessible by all threads and the host CPU [NVI11b]. Additionally, all threads have access to texture and constant memory which are read-only and optimized for specific usages. The global, constant, and texture memories are persistent across multiple kernels for the same application. Additionally, the host (CPU) and device (GPU) memory spaces are separate; copies to and from each context must be made explicitly by the programmer as kernels can only operate on data that has been allocated in device memory. A more detailed description of each type of memory is available in Section 5 of the NVIDIA CUDA C Programming Guide [NVI11b]. Table 2.1 provides a summary of the different memory types [NVI12a] .

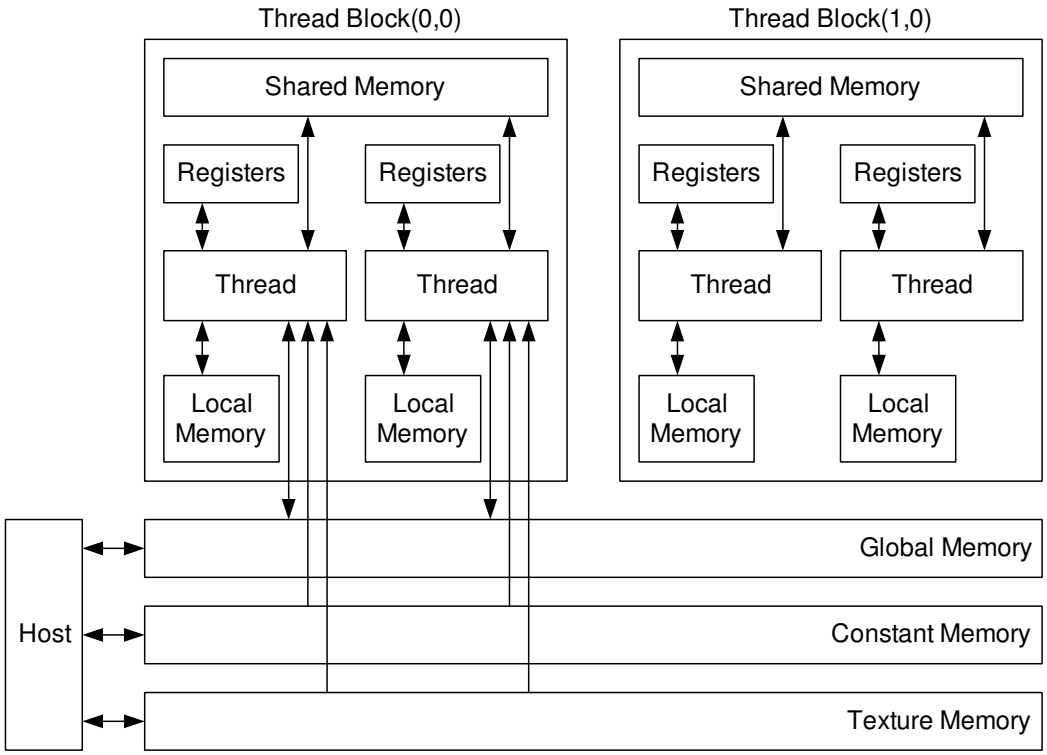

Figure 2.9: CUDA Memory Hierarchy

Table 2.1: CUDA Memory Types

Memory	Location On/Off Chip	Access	Scope	Lifetime
Register	On	R/W	1 thread	Thread
Local	Off	R/W	1 thread	Thread
Shared	On	R/W	All threads in block	Block
Global	Off	R/W	All threads in block	Host allocation
Constant	Off	R	All threads + host	Host allocation
Texture	Off	R	All threads + host	Host allocation

Since CUDA does not cache global memory accesses for all of its hardware versions (compute capabilities), it is important for applications to coalesce reads and writes to memory so as to maximize throughput. Coalescing is the practice of minimizing the total number of transactions made to memory by the warp or half-warp. The techniques used to minimize this differ depending on the compute capability of the CUDA device as later generation devices perform some caching of global and local memory accesses. Section 5 of the NVIDIA CUDA C Programming Guide and Section 6 of the NVIDIA CUDA C Best Practices Guide provide more details [NVI11b][NVI12a].

2.5.4 CUDA Compilation. NVIDIA provides a compiler called NVCC with its CUDA software development kit. As mentioned in Section 2.5.3, CUDA applications are heterogeneous in nature; a portion is compiled for the CPU, and a portion is compiled for the GPU. A standard C/C++ compiler is used to compile the host code, while NVCC compiles the CUDA kernels into assembly files as shown in Figure 2.10 [NVI11b]. The instruction set architecture used by CUDA GPUs is called PTX. It spans multiple GPU generations, allowing for backwards compatibility when run on newer chipsets [NVI11a]. This is accomplished by utilizing just-in-time compilation for the specific chipset. The slowdown of the JIT compilation is partially offset by a cache of compiled PTX created

and managed by the CUDA driver so that future instantiations do not perform redundant compilation. NVCC also produces binary files called cubin files. The advantage of PTX over cubin is the increased cross compatibility between generations of CUDA GPUs; cubin files are compiled for a specific compute capability and will not execute on cards with a different capability version.

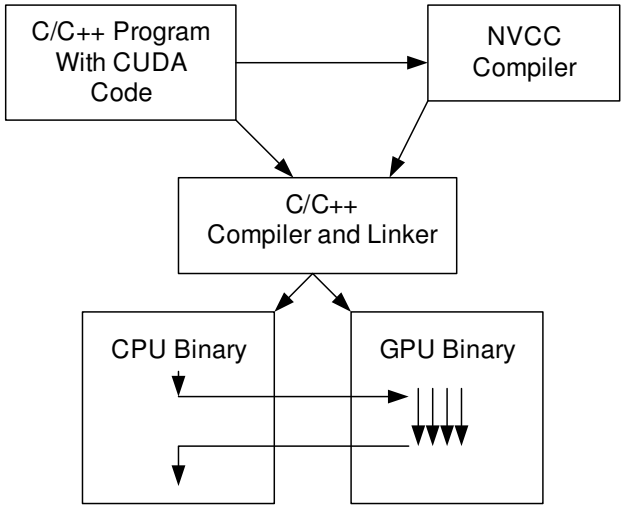

Figure 2.10: CUDA Application Compilation and Linking

2.5.5 CUDA Compute Capability. The device compute capability is a standard used by NVIDIA to indicate which generation a GPU belongs to and its corresponding features. A summary of the features available to different compute capabilities is provided in the NVIDIA CUDA C Programming Guide [NVI11b]. The primary differences in compute capabilities are the allowable dimensions of threads blocks/grids, the number of warps per multiprocessor, and the allowable sizes of various memory spaces. Compute capability 1.3 and higher adds support for double-precision floating point operations, while previous versions only support single-precision floating point operations.

2.6 Related GPGPU Work

This section discusses related GPGPU work with respect to computer and network security and intrusion detection.

The parallel nature of GPGPU is also beneficial to computer and network security applications. GPGPU has shown particular promise in the area of string matching and hash calculation. Kouzinopoulos and Margaritis utilize the GPU to perform string matching using parallel implementations of several commonly used algorithms. They observe a speedup of 24x over the serial CPU implementations [KoM09]. Collange et al. employ the GPU to conduct digital forensics in the area of data carving. The GPU is used to hash byte patterns and search a hash database in GPU memory, realizing a 13x speedup over the CPU [CDD09]. Vasiliadis et at. explore offloading the matching of patterns for network intrusion signatures to the GPU. Their research effort focuses on improving the performance of signature-based IDS, observing a 2-fold speedup over Snort [Sou10][VAP08]. Smith et al. implement a signature pattern matching system using deterministic finite automata and extended finite automata resulting in a speedup of 9x over the CPU implementation [SGO09]. Kovach utilizes CUDA to accelerate the computation of MD5 hashes for detecting malicious files, a technique commonly used in antivirus products such as clamAV, observing speedups of 82% over the CPU implementation [Cla09][Kov10]. Fechner performs similar research using the SHA-1 hashing algorithm [Fec10].

The highly repetitive and computationally expensive nature of many machine learning and data mining techniques lend themselves particularly well to GPGPU implementations. For example, the local outlier factor clustering algorithm is shown by Alshawbkeh et al. to benefit from CUDA parallel execution with a speedup of over 100x as compared to the CPU implementation [AJK10]. Platos et al. demonstrate the potential of the parallel nature of the GPU with a CUDA implementation of a non-negative matrix

factorization-based IDS. They realize maximum speedups of 500x for the training phase and 190x for the testing phase [PKS10]. Zhang et al. use CUDA to perform a feature reduction clustering procedure as a preprocessing step to a CPU support vector machine intrusion detection system. This approach shows an increase in performance of the reduction step of 32x; however, the SVM is still executed on the CPU, reducing the overall gain to 8.28x [ZZW11]. These research efforts illustrate the potential of GPGPU to improve the performance of intrusion detection systems that utilize machine learning for anomaly detection.

2.7 Machine Learning Implementations

This section describes the specific machine learning implementations selected for this research. Section 2.7.1 covers the CPU support vector machine implementation LIBSVM. Section 2.7.2 provides an overview of GTSVM, the GPGPU support vector machine implementation. Section 2.7.3 summarizes the artificial neural network implementation, LIBCUDANN.

2.7.1 LIBSVM. The CPU support vector machine implementation used in this research is LIBSVM Version 3.11. LIBSVM is a C++ library that provides support for multiple types of SVM and kernel functions [ChL11]. LIBSVM employs a sequential minimal optimization (SMO) algorithm to conduct the search for the optimal separators of the training data. Chang and Lin present the basic problem as follows for a generalized form of the support vector classification equations [ChL11]:

$$\min_{a} f(\alpha) \equiv \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q}\boldsymbol{\alpha} + \mathbf{p}^T\boldsymbol{\alpha} \tag{2.10}$$

such that

$$\mathbf{y}^T\boldsymbol{\alpha} = \Delta, \mathbf{0} \leq \alpha_t \leq C, t = 1, \ldots, l,$$

and

$$y_t = \pm 1, t = 1, \ldots, l.$$

where $\mathbf{Q}$ is an l by l semi-definite matrix with $Q_{ij} \equiv y_j y_k K(\mathbf{x_j} \cdot \mathbf{x_k})$ and $K(\mathbf{x_j} \cdot \mathbf{x_k})$, abbreviated K_{jk}, is the kernel function. LIBSVM utilizes a sub-problem optimization method wherein a working set of two elements is used to solve a sub-problem until a stopping point is reached for the algorithm as described in the LIBSVM implementation document [ChL11]. The working set selection process employs a second-order selection heuristic described by Fan et al [FCL05].

2.7.2 GTSVM. The GTSVM library is a support vector machine library developed by the Toyota Technological Institute at Chicago [CSK11]. It provides support for Gaussian, polynomial, and sigmoid kernel support vector machines executed on the GPU using CUDA GPGPU. It uses a SMO-type algorithm similar to that used in LIBSVM. The simplified process is as follows [CSK11]:

1. Choose a working set on the GPU
2. Calculate the Gram matrix restricted to the working set using the CPU and optimize the sub-problem.
3. Update all of the elements in response to the changes using the GPU.

GTSVM uses a working set of 16 elements rather than 2 as in LIBSVM in order to take advantage of the parallel nature of the GPU and minimize the memory accesses needed. Additionally, GTSVM uses a first-order working set selection heuristic, while LIBSVM uses a second-order selection heuristic. The differences between first-order and second-order working set selection heuristics are described by Fan et al [FCL05]. GTSVM attempts to solve:

$$\max_{\alpha \in \Re} \mathbf{1}^T \boldsymbol{\alpha} - \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{Q} \boldsymbol{\alpha} \tag{2.11}$$

subject to: $\forall i(0 \leq \alpha_i \leq C)$ and $\sum_{i=1}^{n} y_i \alpha_i = 0$ with $c_i \equiv \sum_{j=1}^{n} \alpha_j y_j K(x_i, x_j)$ as the responses or support vectors. The working set contains 8 elements maximizing the selection heuristic and 8 elements minimizing it [CSK11].

The calculation of the heuristic values is distributed across the GPU's threads and a parallel max-reduction is performed to locate the maxima. This involves breaking up the list of heuristic values and sorting the chunks in parallel. The maximum values from each chunk are added to a new list and the process is repeated. According to Cotter, this reduction step is one of the most expensive portions of the GPU optimization algorithm. When the results have been reduced to the point that using the GPU is no longer cost-effective due to the small number of values, the CPU is used to find the maxima among the remaining heuristic values. The working set values are then copied to the CPU and the sub-problem is optimized.

After the optimization step, the changes to the variables must be updated in the support vectors. GTSVM performs a clustering of the training vectors by sparsity pattern to allow the memory accesses on the GPU to be coalesced for the working set. The sparsity pattern refers to the presence of zeros in the input data. An example of the clustering is presented in Figure 2.11 [CSK11]. The non-zero values are grouped and processed together by the thread blocks so that the zero values can be ignored, saving execution time and memory accesses. The clustering algorithm operates on the assumption that similar sparsity patterns occur frequently, allowing thread blocks to work on the same cluster pattern for all of the data.

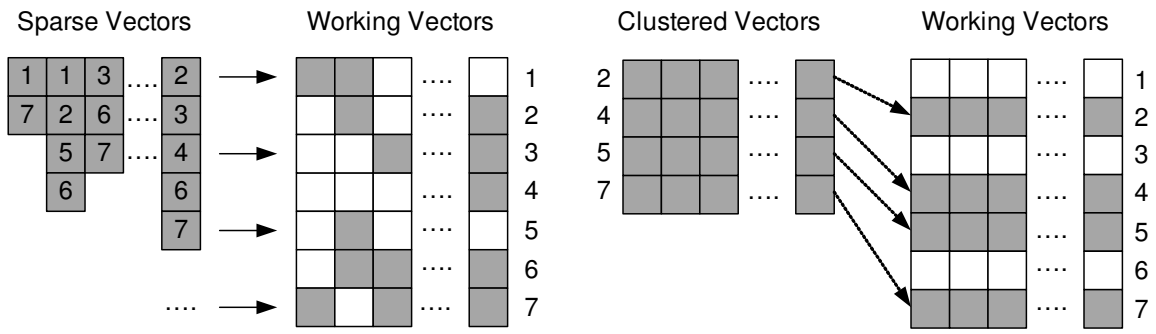

Figure 2.11: GTSVM Clustering Example

2.7.3 LIBCUDANN. This research utilizes LIBCUDANN, a library written by Luca Donati at the University of Parma [Don11]. LIBCUDANN is a C++ implementation of a feed-forward neural network with back-propagation learning. It provides both a CPU implementation and a CUDA GPU implementation. The implementation provides three commonly used activation functions: linear, sigmoid, and hyperbolic tangent. The training is performed either incrementally or in a batch and follows the back-propagation algorithm described in Section 2.3.5.

The parallel implementation utilizes the GPU to perform the activation function and neuron delta calculations for multiple neurons at once. It does this by using the following method:

1. Calculate the neuron outputs for the training instances.
    a. Using CUBLAS (a CUDA linear algebra library) operations, the neuron input values are aggregated and multiplied [NVI12b].
    b. The resulting values are passed to the CUDA activation function kernels to calculate the activation function results for multiple neurons in parallel.

2. The deltas are computed in a similar manner using the resulting neuron outputs and CUDA error function kernels.

3. The network weights are updated by multiplying the neuron values matrix and deltas matrix then adding the result to the previous weights matrix using CUBLAS functions.

Since matrix multiplication and adding is an operation that can be split up into parallel subproblems, it affords a large speedup over a sequential CPU implementation.

2.8 Summary

This chapter provides the background on network intrusion detection using machine learning via CUDA GPGPU. An overview of the history of intrusion detection is given and various intrusion detection techniques are discussed. An overview of machine learning is presented and the underlying theory for support vector machines and artificial neural networks is explored. The NVIDIA CUDA GPGPU programming model and hardware architecture are explored and related work regarding computer security and GPGPU is discussed. Lastly, an overview is given of the specific machine learning implementations selected for this research.

3 Methodology

This chapter presents the goals and methodology of this research. Section 3.1 defines the research goals and approach. Section 3.2 describes the system under test and the scope of the research. Section 3.3 discusses the services provided by the system under test. The workloads offered to the system are discussed in Section 3.4. The metrics used to evaluate the performance of the system are covered in Section 3.5. Section 3.6 describes the system parameters. The factors and levels for the evaluation of the system are discussed in Section 3.7. Section 3.8 describes the evaluation techniques used in this research. The design of the experiments used to evaluate the system under test is presented in Section 3.9. Lastly, a summary of the methodology is provided in Section 3.10.

3.1 Problem Definition

The goals of this research are to:

1. Determine whether using Support Vector Machines (SVMs) or Artificial Neural Networks (ANNs) is the more accurate method for classifying anomalous network traffic on the GPU. This research is limited to the use of Gaussian kernel SVMs and feed-forward, single hidden layer ANNs with batch back-propagation learning and the NVIDIA CUDA GPGPU standard.

2. Evaluate the performance differences between a serial implementation and a parallel implementation of the machine learning intrusion detection system under consideration.

3. Determine which ensemble configuration provides the most accurate method for classifying network payload data using support vector machine and artificial neural network supervised machine learning classifiers. This research is limited to the majority vote and final ANN classifier ensemble combination methods.

The approach used to accomplish these goals is to:

1. Implement a GPU-accelerated, anomaly-based NIDS using the CUDA GPGPU standard to perform the SVM and ANN machine learning. The GTSVM library is used for the GPU SVM classifier [CSK11]. The LIBCUDANN library is used for the GPU ANN classifier [Don11]. An IDS evaluation dataset such as the NSL-KDD dataset creates a baseline for each of the machine learning algorithms under consideration [TBL09b].

2. Implement the anomaly-based NIDS using CPU algorithms for both SVM and ANN. The SVMs under evaluation are Gaussian kernel support vector machines from the LIBSVM library; the LIBCUDANN library provides both a CPU-based implementation and a GPU-based implementation of the artificial neural network under consideration [ChL11][Don11]. The performance metrics of each algorithm are collected for both the parallel and serial implementations of the anomaly-based NIDS.

3. Implement an ensemble of classifiers to classify packet payload data preprocessed using 2_v-gram frequency analysis [PAF09]. Ensembles are created using differing numbers of base classifiers and the majority voting and final ANN classifier combination methods.

3.2 System Boundaries

The System Under Test (SUT) is the GPU-accelerated Network Intrusion Detection System (GNIDS) depicted in Figure 3.1. It consists of a host machine, a traffic preprocessor, a CUDA compatible GPU, a network connection, an anomaly detection method, and an alert reporter. This system receives traffic from the network connection and processes it through its anomaly detector. The detector performs traffic classification and labels the traffic as normal or anomalous. The alert reporter writes a report of

anomalous traffic to a log for future review; it also sends alert summary data for the anomalous connections to an alert monitor. In this research, the scope is limited to the detector itself; the alert monitor is not implemented. Additionally, the GPU chipset scope is limited to GPUs manufactured by NVIDIA for compatibility with CUDA-based machine learning libraries and the host is a Dell T7500 running Windows 7 SP1. This research simulates the SUT's network connection and traffic processor by feeding preprocessed, recorded network traffic to the detection algorithm. The Component Under Test (CUT) is the anomaly detection method. Specifically, the type of machine learning algorithm or ensemble of algorithms in use and whether it is executed on the CPU or GPU is tested.

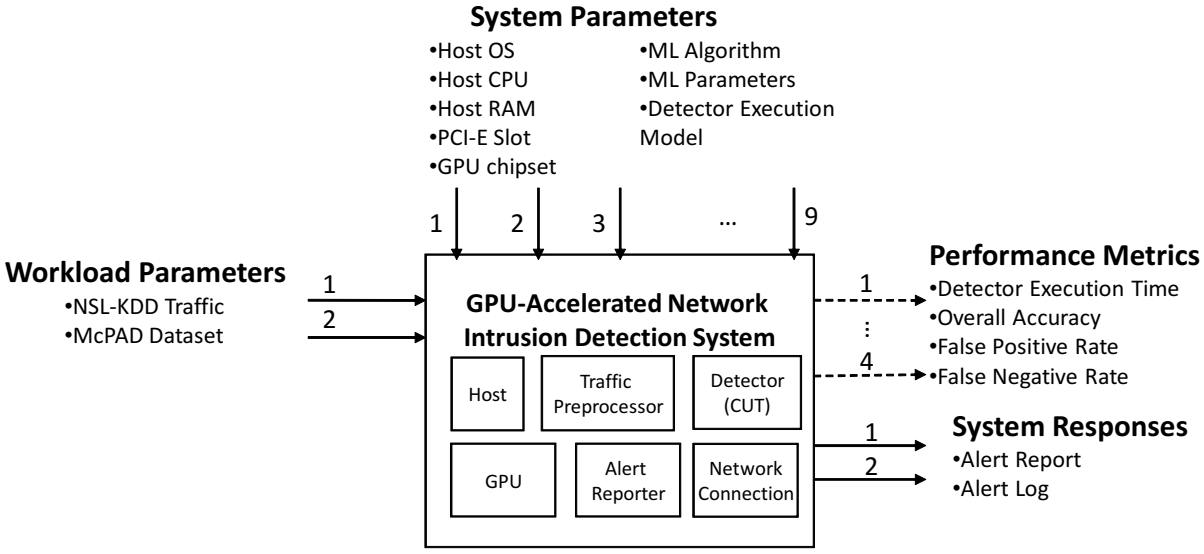

Figure 3.1: GPU-Accelerated Network Intrusion Detection System

3.3 System Services

The service the system provides is the classification of network traffic. The machine learning algorithm used in the anomaly detector analyzes the traffic passed to the system and labels it as one of two classes. The first classification is normal traffic. This is traffic that fits the pattern or baseline of day-to-day traffic for the host or entire network, depending on the detector's assigned scope. When the traffic does not fit the normal pattern of activity, it is labeled as anomalous traffic. Depending on the machine learning algorithm and the quality of the training dataset, traffic can be misclassified. This misclassification can result in normal traffic being labeled as anomalous, a false positive result, and in anomalous traffic being labeled as normal, a false negative result. When an IDS is evaluated for accuracy, these are the primary failure or error conditions considered. As with any computer system, the system can be loaded with more traffic than it can process, resulting in a third failure condition, denial-of-service. The denial-of-service failure modes are outside the scope of this research.

3.4 Workload

The workload offered to the system consists of preprocessed, recorded network traffic. The standard operation for GNIDS is to analyze the network traffic provided over its network connection. For this research, GNIDS is fed network traffic from two test datasets.

The MIT Lincoln Labs DARPA 1998 and 1999 IDS evaluation datasets and their derivative, the ACM Knowledge Discovery and Data mining (KDD)1999 dataset, are commonly used to evaluate anomaly-based IDS performance. The DARPA datasets consist of several weeks worth of recorded traffic from a simulated Air Force network [MIT12]. The KDD-1999 dataset is a processed version of the DARPA 1998 dataset for use by machine learning classifiers [LeS00][ACM99]. This research uses a modified

version of the KDD-1999 dataset, the Network Security Laboratory (NSL) KDD dataset, for the baseline comparison between the considered machine learning algorithms [TBL09b].

The NSL dataset takes the KDD-1999 dataset and attempts to address some of the common complaints against it by modifying the distribution and replication of records in the dataset. The original KDD-1999 dataset has numerous redundant records in both the training and testing data. According to Tavallaee, the training set is reduced by 78.05% by removing the redundant records. These redundant records have the effect of biasing classifiers towards those types of records, resulting in poorer detection rates for the less frequent records [TBL09a]. While this dataset does address some of the problems of KDD-1999, it is still not considered to be representative of network traffic found in a typical enterprise network. However, due to the lack of publicly available IDS evaluation sets, it is used for the purpose of benchmark comparisons between the classifiers in this research.

3.4.1 NSL-KDD Evaluation Workload Creation. This research uses a subset of the NSL-KDD train+.txt file for its evaluation and performs k-fold cross-validation to evaluate the performance of the classifiers [TBL09b][RuN09]. The dataset is split into ten folds. The classifier is trained on nine folds and tested on the last fold. This is repeated ten times using each fold as a test set and the results are collected. The train+.txt file contains 125,973 training examples with the class distribution shown in Table 3.1.

Table 3.1: Traffic Distribution for NSL-KDD Dataset

Traffic Class	Samples	% Total Samples
Normal Traffic	67343	53.458
DoS Traffic	45927	36.457
User to Root Traffic	52	0.041
Remote to Local Traffic	995	0.789
Probe Traffic	11656	9.253

The datasets created for this research consist of randomly selected samples from the NSL-KDD train+.txt file. First, the file is processed and the attack labels are replaced with the traffic class of 0 for Normal traffic, 1 for DoS traffic, 2 for the User to Root traffic, 3 for the Remote to Local traffic, and 4 for the Probe traffic. Since machine learning algorithms such as SVMs and ANNs need numeric inputs, the symbolic features such as the protocol type and flags are mapped to a binary vector of 0s and 1s to indicate which value is present in the instance. For example, the input "TCP" is mapped to $< 0, 1, 0 >$ as it is the second of three values occurring in the dataset. This symbolic substitution is performed for the protocol type (3 values), service (71 values), and flag features (11 values). Additionally, the numeric input features of the file are scaled for each instance according to [ChL12]:

$$X' = \frac{(x - min_i) * (ScaleMax - ScaleMin)}{(max_i - min_i)} \tag{3.1}$$

where X' is the scaled value for a given feature i, x is the original value for the feature i, and max_i and min_i refer to the maximum and minimum values for the feature i in the entire dataset. This research uses a ScaleMin of 0 and a ScaleMax of 1. After scaling the input features, a randomized subset of the dataset is taken, using each instance only once. The subset is created by placing the instances for each traffic class in a list and, using the C# System.Random object, selecting the target number of instances from each list without replacement and placing them in a single output dataset. The original class distribution

and new class distributions (10% subset) are given in Tables 3.1 and 3.2, respectively. The sample labels are mapped to -1 for Attack (classes 1-4) and +1 for Normal (class 0) in the final dataset. The newly created dataset is then split into ten folds for use in k-fold cross-validation [RuN09]. The dataset creation process is repeated for each of the levels of the traffic set size factor as described in Section 3.7. The distributions of the remaining factor levels are included in Appendix A.

Table 3.2: Traffic Distribution for NSL-KDD 10% Subset

Traffic Class	**Samples**	**% Total Samples**
Normal Traffic	6734	53.465
DoS Traffic	4592	36.459
User to Root Traffic	5	0.04
Remote to Local Traffic	99	0.786
Probe Traffic	1165	9.25

3.4.2 McPAD Evaluation Workload Creation. The second workload offered to the system is a set of traffic payloads provided by Perdisci [Per09]. The normal data consists of a subset of the normal traffic from the first week of the 1999 DARPA evaluation dataset. The attack data consists of several standard HTTP attacks and several capture files of polymorphic blending attacks designed to evade payload frequency analysis based techniques like those used in Multiple-Classier Payload-based Anomaly Detector (McPAD) and this research [PAF09]. The files used in this research are provided with McPAD. The payloads are processed using the variation on n-gram frequency analysis implemented by Perdisci for McPAD. Perdisci utilizes a sliding window variant of 2-gram analysis called 2_ν-gram analysis [PAF09]. The occurrence frequency of bytes ν positions apart from each other is measured. A sliding window of length ν+2 is used to measure all

of the byte pairs in the payload. By using distinct values of ν, different characteristics of the payloads are modeled by the occurrence frequencies. A clustering algorithm is used to reduce the number of features resulting from the 2_ν-gram process. The feature clustering process results in a dataset containing a set of clustered features; in this research, 40 clusters and 160 clusters are used.

McPAD utilizes unsupervised machine learning to perform its classification, requiring only non-malicious traffic for training. McPAD is trained using preprocessed non-malicious traffic and uses its 2_ν-gram packet processor to process and classify test packets at runtime. Since this research focuses on supervised learning instead of unsupervised and uses preprocessed files instead of implementing the real-time packet processor, the McPAD processing technique is used to preprocess all of the evaluation traffic. As in McPAD, the training.pcap traffic packets are used to create a set of feature clustering maps. The maps are then used to create the processed versions of both the non-malicious and malicious traffic pcap files. This results in a set of files containing the preprocessed instances and labels for each input pcap file. The processed files are then combined into a single dataset. Due to the size of the training.pcap file, a 25% subset of randomly selected instances is used in the creation of the combined dataset; all of the other files are used in their entirety. For repeatability, the specific packet instances used in each file are recorded. A randomized 60% subset of the dataset is selected, using each instance only once and is split into ten folds for use in cross-validation. As with the NSL-KDD workload, a C# program is used to perform the random selection by creating lists of the instances for each traffic file and selecting, without replacement, the target number of instances from each list using the C# System.Random object. The subset distribution is presented in Table 3.3. This process is repeated for all seven values of ν used in this research as described in Section 3.7.

Table 3.3: Input File Distribution for McPAD 60% Subset

Input File	**Instances**	**% of Dataset**
training (25% subset)	15327	71.18%
1-gram-attacks-all	1878	8.72%
2-all-gram-attacks-all	1869	8.68%
all_attacks_payloads	121	0.56%
all_morphed_shellcode_attacks_payloads	475	2.21%
12-gram-attacks-all	1863	8.65%

3.5 Performance Metrics

The following metrics are used to evaluate the performance of GNIDS:

- Detector Execution Time - The execution times for the training and testing operations are collected for the anomaly detector. These times are used to evaluate the relative performance of the CPU and GPU implementations for the machine learning algorithms under consideration.

- Overall Accuracy - The overall accuracy for each classifier is collected. It is calculated as the number of true positives and true negatives divided by the total number of instances analyzed. The system determines the number of true positives and true negatives by comparing the predicted traffic labels to the labels provided with the validation folds. The accuracies for each cross-validation run are averaged and used as the overall accuracy for the experimental run [RuN09].

- False Positive Rate - For an anomaly-based intrusion detection system, any traffic classified as an anomaly that is actually valid traffic is a false positive. The false positive rate is the proportion of negatives that are incorrectly labeled as positives. The false positive rate is calculated as the number of false positives divided by the

number of false positives and true negatives [Ham12]. The system determines the number of false positives and true negatives by comparing the predicted traffic labels to the labels provided with the validation folds. The false positive rates for each cross-validation run are averaged and used as the overall false positive rate for the experimental run [RuN09].

- False Negative Rate - For an anomaly-based intrusion detection system, any traffic classified as normal traffic that is actually anomalous traffic is a false negative. The false negative rate is the proportion of positives that are incorrectly labeled as negatives. The false negative rate is calculated as the number of false negatives divided by the number of false negatives and true positives [Ham12]. The system determines the number of false negatives and true positives by comparing the predicted traffic labels to the labels provided with the validation folds. The false negative rates for each cross-validation run are averaged and used as the overall false negative rate for the experimental run [RuN09].

3.6 System Parameters

The parameters for the system under test are:

- Host Operating System - The OS impacts the compatibility level of the GPU drivers with the GPGPU standards. Windows 7 SP1 64-bit is used as it is the newest Windows OS currently supported.

- Host CPU - The CPU chipset and performance characteristics can impact the performance of both the CPU and GPU implementations of the detection system as portions of GPGPU programs are executed on the host CPU. The host is a Dell T7500 configured with dual Xeon X5650 processors running at 2.67 GHz.

- Host RAM - The amount of host system memory impacts the performance of the CPU and GPU implementations of the detection system as the host RAM is used for portions of both implementations. The test system used in this research is equipped with 48 GB of DDR3 memory.

- Host Hard Drive - The hard drive of host system impacts the performance of the CPU and GPU implementations of the detection system as the host performs swaps of host memory with the page file. The Dell T7500 used in this research is equipped with 2 7200 rpm 1 TB drives in a RAID 0 configuration.

- PCI-Express Slot - The version of the PCI-E slot in the host can limit the performance of the SUT since older versions of the PCI-E standard cannot support the higher data throughput required by newer GPU chipsets. The PCI-E version used in this research is version 2.0.

- GPU Chipset - The specific chipset of the GPU has performance implications for GPGPU applications. Some chipsets have substantially fewer processing cores than others as well as a wide range of GPU clock speeds. The amount of video memory on the specific GPU also limits the amount of traffic data that can be stored and analyzed by the GPGPU detector. Additionally, the bandwidth of the video memory varies from GPU to GPU which impacts how quickly data can be transferred. This research uses a PNY NVIDIA GeForce GTX 280 graphics card. The GTX 280 is a CUDA compute capability 1.3 device with 30 multiprocessors and 240 CUDA cores [NVI11b]. The GTX 280 is used as it has the same compute capability as the card used in the development of GTSVM, the CUDA support vector machine library selected for this research. The key hardware specifications of the card are presented in Table 3.4 [PNY12].

Table 3.4: Specifications for PNY GeForce GTX 280

Component	**Specification**
Bus Type	PCI-E 2.0
Core Clock	602 MHz
Shader Cores	240
Shader Clock	1296 MHz
Memory bus	512-bit
Memory Size	1024 MB DDR3
Memory Clock	2214 MHz
Memory Bandwidth	141.7 GB/s

- Machine Learning Algorithm - The particular detection algorithm has the largest potential impact on the performance of GNIDS. The training time, classification time, and overall accuracy are all directly dependent on which algorithm is in use.

- Algorithm Parameters - The parameters used to create the machine learning classifier directly affect the execution time and classification performance of the given algorithm. The algorithm parameters (cost and gamma for SVMs and hidden layer size for ANNs) are chosen based on a preliminary parameter search performed on a separate dataset. The parameters are described in Section 3.8.

- Detector Execution Model - The execution model used for the anomaly detection algorithm directly impacts the performance of GNIDS. The parallel model executes the algorithms using CUDA GPU libraries, while the serial model utilizes CPU implementations of the algorithms.

3.7 Factors

The factors and levels for this research are either system factors or workload factors. A summary of the factor levels are given in Tables 3.5 and 3.6 for the NSL-KDD and McPAD workloads, respectively.

Table 3.5: Factors and Levels for NSL-KDD Workload

Factor	**Level 1**	**Level 2**	**Level 3**
Machine Learning Algorithm	ANN	SVM	—
SVM Parameter Set	High-C	Low-C	—
Detector Execution Model	CPU	GPU	—
NSL- KDD Traffic Set Size	10% Subset	20% Subset	30% Subset

Table 3.6: Factors and Levels for McPAD Workload

Factor	**Level 1**	**Level 2**	**Level 3**
Machine Learning Algorithm	ANN	SVM	—
Detector Execution Model	CPU	GPU	—
Number of Base Classifiers	3 BCs	5 BCs	7 BCs
Number of Feature Clusters	40 Clusters	160 Clusters	—
Combination Method	Majority Vote	Final ANN Classfier	—

3.7.1 System Factors. The system factors for this research are:

- Machine Learning Algorithm - A primary goal of the research is to evaluate the impact of using different machine learning algorithms for anomaly-based detection. GNIDS is tested with a Gaussian kernel support vector machine and a feed-forward,

single-layer neural network with batch back-propagation learning as these are two commonly used machine learning algorithms that have CUDA implementations.

- Detector Execution Model - An evaluation of the performance difference afforded by implementing the intrusion detection system in a CUDA GPU parallel model rather than a serial CPU model is a primary goal of this research. The levels for this factor are CUDA GPU parallel and CPU serial. It is anticipated that the use of the GPU model will have a large impact on the execution time of the detection system, especially in the classifier training phase.

- Number of Base Classifiers - The number of base classifiers used in an ensemble classifier contributes to the amount of information used to make the ensemble's final decision for the traffic. This factor has three levels: 3 base classifiers, 5 base classifiers, and 7 base classifiers. Each base classifier is given a dataset processed using a different value of ν. The specific values of ν used to train the base classifiers are presented in Table 3.7.

Table 3.7: Values of ν for Base Classifier Datasets

Base Classifiers	**Values of ν Used**
3 base classifiers	3, 5, 7
5 base classifiers	1, 3, 5, 7, 9
7 base classifiers	0, 1, 3, 5, 7, 9, 10

- Ensemble Combination Method - The ensemble combination method is used by the ensemble to decide which label to give a traffic instance based on the predictions of the base classifiers. This factor has two levels: majority vote and final ANN classifier. The final ANN classifier operates by using the predictions of the base classifiers for the training data as training input for an ANN classifier to determine

dynamic weights for each base classifier. The final classifier factor level adds significant processing time over the majority vote method, but has the potential to increase the overall accuracy due to its ability to dynamically weight the base classifier predictions.

3.7.2 Workload Factors. The workload factors for this research are:

- NSL-KDD Machine Learning Algorithm Parameters - The machine learning parameters used for the KDD workload are chosen based on a preliminary parameter search using the provided KDDTrain+20 Percent file [TBL09b]. The ANN use a hidden layer of 32 neurons for this workload. Two parameter sets are used for the SVM classifiers. The high-c set uses a value of 64 for C and a value of 2 for Gamma. The low-c set uses a value of 1 for C and a value of 8 for Gamma. These sets are chosen to evaluate the performance of the SVM implementations with both high and low cost values while maintaining comparable accuracies. Both SVM implementations are evaluated with both the high-c and low-c parameter sets.

- NSL-KDD Traffic Set Size - The size of the dataset used to train and test the machine learning classifiers directly impact the training and testing time of the classifiers. This factor has three levels: a 10% subset of the train+.txt file, a 20% subset, and a 30% subset. Each subset is sampled from the same population of data and is made up of unique samples from the NSL-KDD dataset as described in Section 3.4.

- McPAD Ensemble Machine Learning Algorithm Parameters - The machine learning parameters for the McPAD workload are chosen based on a parameter search using a separate subset of the combined McPAD dataset. The ANNs use a 2 neuron hidden layer for this workload. The CPU SVM uses a C of 64 and the GPU SVM uses a C of 0.125. Gamma is fixed at 0.5 for both SVMs. These values of C are

chosen to evaluate the relative performance of the SVM implementations on the McPAD dataset while maintaining comparable accuracies.

- McPAD Feature Clusters - A primary goal of this research is to determine performance differences between the CPU and GPU implementations of the ensemble classifier on 2_v-gram processed payload data. This factor has two levels: the packet preprocessor is run with 40 clusters and 160 clusters to determine the impact of including more input feature information on the performance of the base classifiers and the accuracy of the ensemble as a whole. The values of 40 and 160 are chosen as these are the median and maximum number of clusters Perdisci uses in the evaluation of McPAD [PAF09].

3.8 Evaluation Technique

The evaluation technique for this research is direct measurement. The detector records system performance counter values before and after running the machine learning training and testing operations and calculates the number of microseconds that elapsed. The timer methods used to record these values are the QueryPerformanceCounter and QueryPerformanceFrequency functions provided by windows.h. When the timer is started, it records the value of the counter in an internal variable and the second call returns the number of microseconds elapsed since the timer was started. For the sake of simplicity, the traffic data is preprocessed for the respective detection algorithms and read from a file instead of transmitted over the network and examined using a packet capture tool.

3.9 Experimental Design

This research uses a full factorial experimental design. Since the research goals can be separated into two categories, the performance of the detector implementations and the

performance of the ensemble configurations, the research consists of two full factorial experiment sets.

The first set of experiments determines the effect of the machine learning algorithm and the execution model on the performance of the detector. The experiment is run with each of the two machine learning algorithms for each of the execution models with both parameter sets for SVM and the three file sizes. This gives six algorithm levels and three dataset sizes for a total of 18 experiments before considering replications. Support vector machines are deterministic for the same kernel parameters and training data so the variance is not significant for the accuracy metrics. The execution time, however, is dependent on concurrent processes on the test machine and the kernel parameters used to train the classifier. Artificial neural networks are trained with a random initialization, resulting in non-deterministic output for the same training data and layer parameters.

The second set of experiments determines the relative performance of the machine learning classifiers on payload analysis data. Ensembles are trained utilizing each machine learning algorithm with each execution model for two feature cluster sizes and three ensemble sizes for both of the ensemble combination methods. This gives a total of 48 experiments before considering replications.

Each factor level evaluation is conducted using 10-fold cross-validation. This results in 180 training and testing operations for the first experiment and 480 for the second before considering replications. Due to the large number of folds and factor levels, only five replications are conducted to account for variance in execution times and accuracies for both experiments. This gives a total of 330 experiments. As a result of the low number of replications conducted, the 90% confidence level is used for the statistical analysis of the results.

3.10 Methodology Summary

This research explores the performance implications of using machine learning algorithms to perform anomaly-based intrusion detection in a parallel computing context. The factors identified for evaluation are the specific machine learning algorithm used, the execution model for the detection method, and the ensemble configuration. The performance metrics chosen to evaluate the impact of these factors are the false positive rate, the false negative rate, the detector execution time, and the accuracy of the detector. A full factorial experimental design is used on an actual implementation, resulting in a total of 330 experiments.

4 Results and Analysis

This chapter presents and analyzes the results of the two experiments. The results of Experiment 1 are discussed in Section 4.1. Section 4.2 presents the results and discussion of Experiment 2. An overall analysis is presented in Section 4.3. Lastly, a chapter summary is given in Section 4.4.

4.1 Results and Analysis of Experiment 1

In Experiment 1, the NSL-KDD dataset is used to create a baseline comparison of the relative performances of the CPU and GPU machine learning algorithms under consideration. The size of the datasets is varied for all of the machine learning algorithm factor levels. The relative performance of each of the classifiers is examined at each of the dataset sizes. Using version 2.13.1 of the R statistical software package for Windows, Tukey's Honest Significant Difference test is used to identify the significant differences in mean execution times at the 90% confidence level [Tea12]. The accuracies of the classifiers are compared for each of the factor levels using Wilcoxon's Signed-Rank test using R and the coin package [HHV08]. Section 4.1.1 examines the relative performances of the CPU and GPU implementations of the machine learning classifiers. Section 4.1.2 examines the relative accuracies of the machine learning classifiers.

4.1.1 Analysis of Differences in Execution Time. This section examines the nature of the performance differences for the training and testing operations of the CPU and GPU classifiers. Section 4.1.1.1 presents the results of the Tukey tests for the differences in mean execution times between classifiers. Section 4.1.1.2 compares the performances of the ANN implementations. Section 4.1.1.3 evaluates the impact of the SVM parameters on the performance of the CPU and GPU implementations. An evaluation of the performance differences between the CPU and GPU SVMs is conducted in Sections 4.1.1.4 and 4.1.1.5.

4.1.1.1 *Statistical Significance of Differences in Execution Time.*

Tukey's Honest Significant Difference (HSD) test is used to compare the 90% confidence intervals for the classifier's average train and test times. The results of the Tukey tests indicate that the size of the dataset has a significant impact on the training and testing times for each algorithm at each factor level. None of the 90% confidence intervals for the difference in mean execution times included zero.

A comparison between the classifier types at each of the dataset size factor levels shows there is a significant difference in the mean training times between all of the classifiers. When comparing the testing times of the algorithms, most comparisons show a significant difference in mean testing times for each of the size factor levels. At the larger size factor levels, however, the high-c GPU (GH) support vector machine and the low-c GPU (GL) support vector machine do not show a significant difference in mean testing times. This occurs at the 20 percent and 30 percent factor levels as shown in Table 4.1. The table presents the results of the Tukey HSD test for the comparison of mean testing times of the two GPU SVMs. The difference column presents the difference in the mean testing time for the 20 and 30 percent traffic size factor levels. The Lower and Upper columns present the lower and upper limits of the 90% confidence interval for the difference in means. As the intervals for both factor levels include zero, there is no significant difference in mean testing time for the GPU SVMs at the 90% confidence level. The rest of the classifiers do show a significant difference in mean testing times. The R commands used for the Tukey tests are included in Appendix A.

Table 4.1: Difference in Mean Testing Time (ms) for the GPU Support Vector Machines

Factor Level	**Algorithms**	**Lower**	**Difference**	**Upper**	**Adjusted P-Value**
20 Percent	GL-GH	-5.17196	1.99582	9.16360	0.9703667
30 Percent	GL-GH	-13.68219	5.07120	23.82459	0.9738601

4.1.1.2 Comparison of ANN Implementations. Tables 4.2 and 4.3 provide the mean training and testing times for the ANN implementations for each of size factor levels. An examination of the ratios of the execution times provides an indication of the degree of difference between the mean execution times of the CPU and GPU implementations for the same machine learning algorithm. The GPU ANN (AG) achieves an average training time speedup of 27.55x over the CPU ANN (AC) implementation for the 10 percent dataset. As shown in Table 4.4, the larger datasets show larger speedups with averages of 28.51x and 29.17x, respectively. The GPU ANN achieves an average testing time speedup of 2.46x over the CPU ANN for the 10 percent dataset and speedups 3.72x and 4.53x for the larger datasets as shown in Table 4.4. The increasing trend in the speedups are easily explained by the nature of the calculations involved and the observed impact of the size of the dataset on the CPU and GPU implementations.

Table 4.2: Mean Execution Times for CPU ANN (AC)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	260.820	0.410	19.131	0.038
20	525.340	0.620	38.080	0.140
30	800.060	0.430	56.901	0.168

Table 4.3: Mean Execution Times for GPU ANN (AG)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	9.466	0.006	7.767	0.060
20	18.429	0.001	10.251	0.093
30	27.424	0.004	12.561	0.095

Table 4.4: Ratio of CPU ANN Times to GPU ANN Times (AC/AG)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	27.554	0.057	2.463	0.018
20	28.506	0.035	3.715	0.034
30	29.174	0.019	4.530	0.036

In the training phases, an ANN must iterate through the entire back-propagation algorithm again for each added training example, a linear increase in complexity. The testing operation is similar. The ANN must calculate the aggregate inputs for each new example and compute the activation function values for each neuron in the network. As long as the number of neurons in the network stays the same, the increase in the number of calculations is directly proportional to the increase in the size of the testing data. As a result, the serial implementation of the CPU is affected in an approximately linear fashion for both training and testing as shown in Figures 4.1 and 4.2. The GPU, however, can perform these added calculations in parallel, achieving better scaling ratios for its execution times. In the training phase, the GPU scaling ratio is approximately linear as shown in Figure 4.1. The testing phase, however, shows superior scaling for the GPU as shown in Figure 4.2.

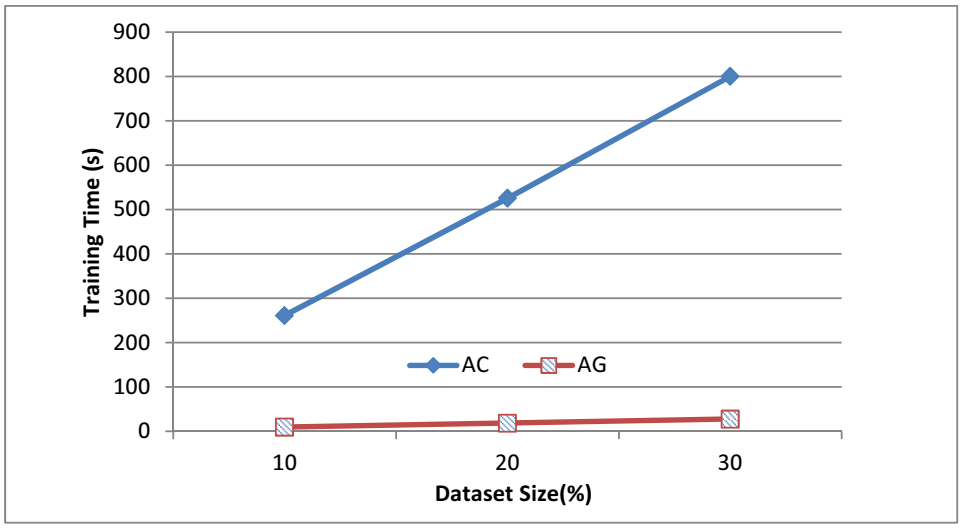

Figure 4.1: ANN Training Times versus Dataset Size

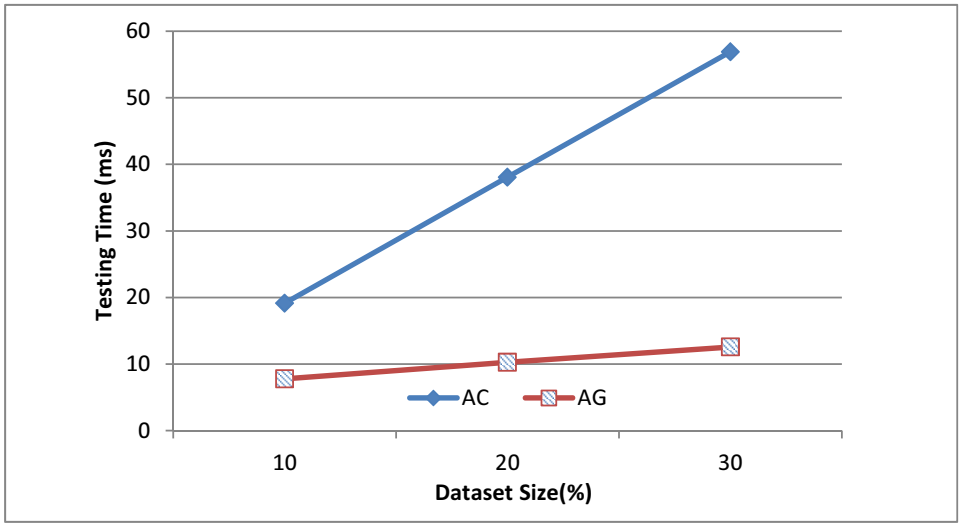

Figure 4.2: ANN Testing Times versus Dataset Size

The difference in dataset size impact between the GPU train and test times is likely the result of the difference in complexity of the training and testing operations. The training step requires the execution of a CUBLAS call and a CUDA kernel for the feed-forward and back-propagation steps and a CUBLAS call for the network weights update before copying the values to host memory. Each of these steps is dependent on the

previous steps in the back-propagation algorithm. The testing operation, however, requires fewer operations before the resulting values are copied to host memory. Additionally, the calculated values are independent, meaning that more examples can be calculated in parallel without waiting for results of previous calculations.

4.1.1.3 Impact of Parameter Sets On SVM Implementations. A comparison of the training and testing times for the CPU SVM using both the high-c (LH) and low-c (LL) parameter sets provides an indication of the impact of changing the C parameter on the performance of LIBSVM for this dataset. Tables 4.5 and 4.6 provide the mean training and testing times for the CPU SVM implementation for each parameter set. For each factor level, the high-c CPU outperformed the low-c CPU in terms of training time as shown in Table 4.7.

Table 4.5: Mean Execution Times for CPU SVM-High (LH)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	5.239	0.016	336.205	0.441
20	28.170	0.140	958.340	5.630
30	95.830	0.620	1801.910	6.660

Table 4.6: Mean Execution Times for CPU SVM-Low (LL)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	12.013	0.044	649.047	0.334
20	90.330	0.310	2219.080	7.730
30	188.100	1.200	4869.700	24.200

Table 4.7: Ratio of CPU SVM-L Times to CPU SVM-H Times (LL/LH)

Size	Training	90% C.I.	Testing	90% C.I.
10	2.293	0.006	1.931	0.002
20	3.207	0.026	2.316	0.020
30	1.963	0.020	2.703	0.021

The average training speedups for the 10 percent dataset and 30 percent dataset are 2.3x and 1.96x, respectively. The 20 percent speedup was greater at an average of 3.2x. This greater value for the 20 percent set is due to the relative effects of scaling on the classifiers. As is shown in Figure 4.3, the high-c SVM is less impacted by the increase from 10 to 20 than the low-c. The 20 to 30 step, however, had a more severe impact on the high-c (LH) than the low-c (LL) as indicated by the larger increase in slope of the high-c from 20 to 30. The slope of the low-c increased by approximately 25%, whereas the slope of the high-c nearly tripled.

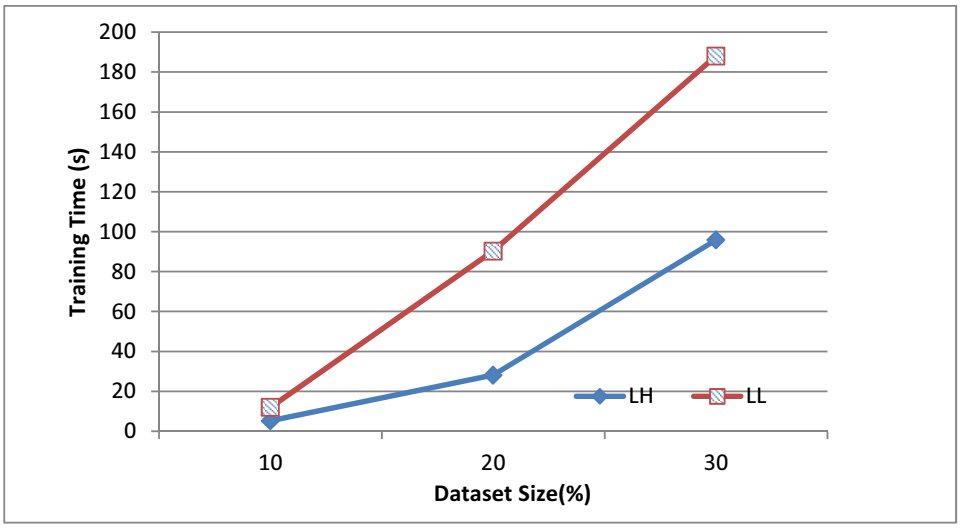

Figure 4.3: CPU SVM Training Times versus Dataset Size

When comparing testing times, the high-c CPU SVM also performs consistently better than the low-c CPU SVM with average testing speedups of 1.93x, 2.32x, and 2.7x for the three dataset levels as shown in Table 4.7. Unlike the training phase, the scaling for the high-c SVM in the testing phase is superior to that of the low-c as shown in Figure 4.4. These results indicate that the CPU SVM implementation shows better performance when higher cost and lower gamma parameters are used for the NSL-KDD workload.

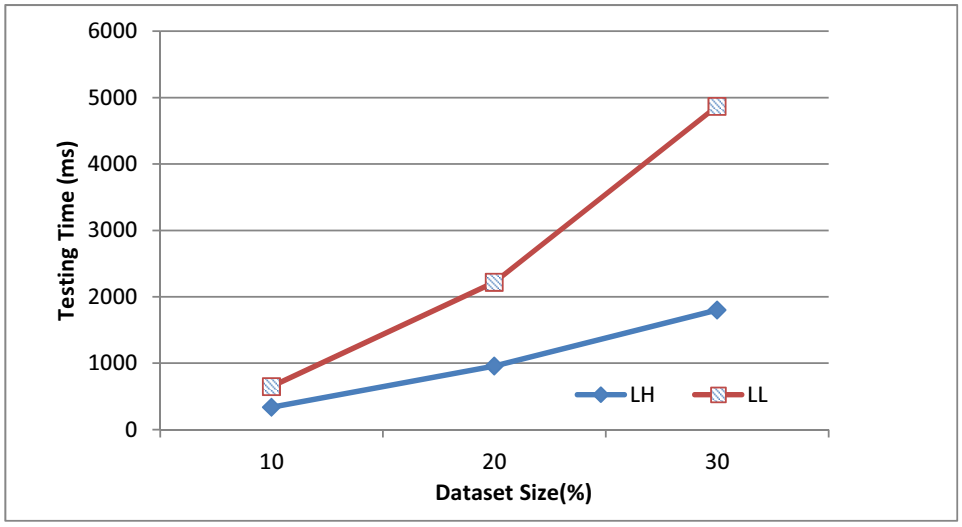

Figure 4.4: CPU SVM Testing Times versus Dataset Size

On the GPU, the low-c SVM (GL) performs consistently better than the high-c SVM (GH) when comparing the training times as shown in Figure 4.5. For the 10 percent dataset, the average speedup is 8.83x. The 20 and 30 percent datasets show greater speedups at averages of 10.6x and 13.6x as shown in Table 4.8. Unlike the CPU SVMs, the differences in scaling between the GPU SVM classifiers are not as large as shown in Tables 4.9 and 4.10; however, the high-c SVM shows a greater increase in relative execution time than the low-c SVM, leading to the increasing gains for the larger datasets.

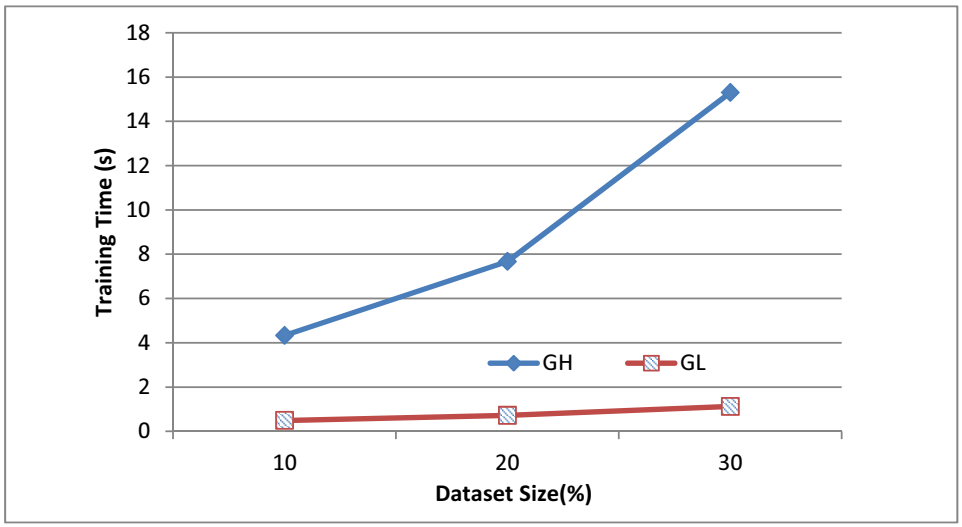

Figure 4.5: GPU SVM Training Times versus Dataset Size

Table 4.8: Ratio of GPU SVM-H Times to GPU SVM-L Times (GH/GL)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	8.833	0.400	0.964	0.033
20	10.589	0.262	0.973	0.010
30	13.613	0.617	0.952	0.014

The two GPU support vector machines show comparable testing times with the high-c slightly outperforming the low-c as shown in Table 4.8. These differences are only statistically significant at the 10 percent factor level as is shown by the Tukey test results discussed in Section 4.1.1.1. Tables 4.9 and 4.10 provide the mean training and testing times for the GPU SVM implementation for each parameter set. These results indicate that the GPU SVM implementation shows better training and comparable testing performance when lower cost parameters are used for this workload.

Table 4.9: Mean Execution Times for GPU SVM-High (GH)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	4.330	0.150	41.810	1.230
20	7.670	0.150	72.322	0.719
30	15.300	0.310	100.361	0.785

Table 4.10: Mean Execution Times for GPU SVM-Low (GL)

Size	**Training Time (s)**	**90% C.I.**	**Testing Time (ms)**	**90% C.I.**
10	0.490	0.007	43.389	0.443
20	0.725	0.017	74.318	0.421
30	1.125	0.032	105.430	1.130

Overall, the SVM testing times are impacted less by the increase in dataset size than the training times. This is attributable to the relative difficulty of the classification operation to the training operation. To classify a new example, the SVM applies the previously computed decision function to the instance, computing a sum of the product of support vectors and the kernel function value of the new example's input features. In the training phase, each example is evaluated to determine the optimal support vectors to separate the training examples with a maximum separating margin according to the cost and gamma parameters used, a significantly more complicated operation. Like the ANN implementations, the GPU SVM shows better scaling capabilities than the CPU SVM.

4.1.1.4 Comparison of CPU and GPU SVM Training Times. The parallel nature of the GPU implementation is evident in the observed speedups of the GPU SVMs over the CPU SVMs. Both of the GPU SVMs outperform the CPU implementation on all of the dataset sizes. The performance difference between the GPU and CPU implementations increases as the size is increased because the execution times on the

GPU are not impacted as severely by the size increase as is discussed in Section 4.1.1.3. Since the GPU performs the training of the SVM using blocks of threads, the additional work is divided up among more blocks. This effectively increases the efficiency of the GPU as more training examples are added, whereas the CPU implementation must perform all of its additional work serially. The ratios of execution times for the support vector machine implementations are given in Tables 4.11-4.14.

The high-c GPU SVM and high-c CPU SVM are fairly close in training performance for the smallest dataset with an average gain of 1.2x for the GPU implementation. At the larger sizes, however, the superior scaling of the GPU implementation results in performance gains of 3.7x and 6.3x over the CPU SVM (Table 4.11). The gains over the low-c CPU also show increasing gains as the dataset size is increased, ranging from 2.8x to 12.3x for the different sizes (Table 4.12).

Table 4.11: Ratio of CPU SVM-H Times to GPU SVM-H Times (LH/GH)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	1.212	0.043	8.047	0.233
20	3.674	0.069	13.252	0.083
30	6.267	0.127	17.955	0.108

Table 4.12: Ratio of CPU SVM-L Times to GPU SVM-H Times (LL/GH)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	2.779	0.096	15.535	0.448
20	11.781	0.217	30.690	0.410
30	12.301	0.276	48.525	0.551

The low-c GPU SVM shows much larger differences in performance when compared to the CPU implementations. As shown in Table 4.13, due to the significantly lower impact of the dataset size on the GPU low-c SVM, the gains increase dramatically as the dataset size is increased. The low-c CPU comparison shows even larger gains with an average of 167x for the largest dataset as shown in Table 4.14.

Table 4.13: Ratio of CPU SVM-H Times to GPU SVM-L Times (LH/GL)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	10.689	0.139	7.749	0.078
20	38.898	0.765	12.895	0.080
30	85.250	2.420	17.092	0.193

Table 4.14: Ratio of CPU SVM-L Times to GPU SVM-L Times (LL/GL)

Size	**Training**	**90% C.I.**	**Testing**	**90% C.I.**
10	24.511	0.374	14.960	0.155
20	124.743	3.207	29.860	0.217
30	167.337	5.210	46.193	0.618

4.1.1.5 Comparison of CPU and GPU SVM Testing Times. As is discussed in Section 4.1.1.3, the CPU SVM implementation shows considerably more impact on the testing time from the increase in traffic set size than the GPU SVM implementation. As with the training time ratios, this results in an increasing speedup over the CPU implementation as the size is increased.

- The high-c GPU shows average gains over the high-c CPU of 8x, 13.3x, and 18x for the three dataset sizes as presented in Table 4.11.

- The high-c GPU shows average gains over the low-c CPU SVM of 15.5x, 30.7x, and 48.5x for the three dataset sizes as shown in Table 4.12.
- The low-c GPU shows similar gains over the CPU implementations as given in Tables 4.13 and 4.14.

These performance differences are easily explained by the nature of the computations involved. Since the classification of a test set by a support vector machine requires the same calculations for each example, the GPU can launch additional blocks of parallel threads to handle the increased number of instances, while the CPU must compute the classification of each example in series.

This section evaluates the performance differences between the serial and parallel implementations of GNIDS. Overall, both of the GPU machine learning algorithms are less affected by increases in workload than the CPU implementations. For the support vector machines, the GTSVM library is significantly faster than the LIBSVM library for both parameter sets, especially for the larger traffic set sizes. Additionally, it is observed that the CPU SVM library does not scale as well with high-c parameters as it does with low-c parameters; however, the performance is significantly better for the high-c CPU SVM at each size level. The GPU SVM library performs better with lower ranging cost parameters in both scaling and training time, but shows little difference in testing time.

4.1.2 Analysis of Classifier Accuracy. This section evaluates the overall prediction accuracies for the machine learning algorithms to determine the more effective algorithm for use on the GPU and the differences in accuracy between the CPU and GPU implementations. Due to the deterministic nature of support vector machines, the same input parameters and training datasets will result in the same accuracy for each replication. As a result, a non-parametric test is used to determine if there is a significant difference between the accuracies at the various factor levels. The accuracies of the

classifiers are compared for each of the factor levels using Wilcoxon's Signed-Rank test in R. The specific R commands used are included in Appendix B. Due to the few number of replications conducted, the 0.10 significance level is used.

An analysis of the differences in results between the CPU and GPU implementations of each machine learning algorithm for each dataset provides an indication of the relative accuracies of the implementations. The Wilcoxon test indicates that there is no significant difference between the CPU (AC) and GPU (AG) artificial neural network implementations for any of the datasets. This is as expected since the implementations use the same algorithm regardless of the computing device.

The p-values for each of the other comparisons indicate there is a significant difference between the other classifiers for each of the datasets. This result is as expected for the low-c and high-c SVM comparisons since the changes in cost and kernel parameters for a support vector machine will impact the values for the underlying support vectors. Changing the cost parameter impacts how sensitive the support vector machine is to misclassified points when optimizing its support vectors and changing the value of the gamma parameter for the Gaussian kernel impacts the overall shape and flexibility of the margin separator, affecting the classifier's ability to separate the training examples [HuW10]. The differences between the CPU and GPU implementations with the same SVM parameters are likely due to the differences between the LIBSVM and GTSVM support vector optimization algorithms, causing the support vector machines to arrive at different solutions for the separator of the training data. The specific R commands used are included in Appendix B.

At each of the traffic set size factor levels, the SVM classifiers are significantly more accurate than the artificial neural network classifiers. Figure 4.6 presents the average accuracies for the classifiers. Of the SVM classifiers, the high-c GPU SVM (GH) has the highest accuracy; the low-c CPU SVM (LL) has the lowest. The low-c GPU SVM (GL)

shows higher accuracies than the low-c CPU SVM, but lower than both of the high-c SVMs.

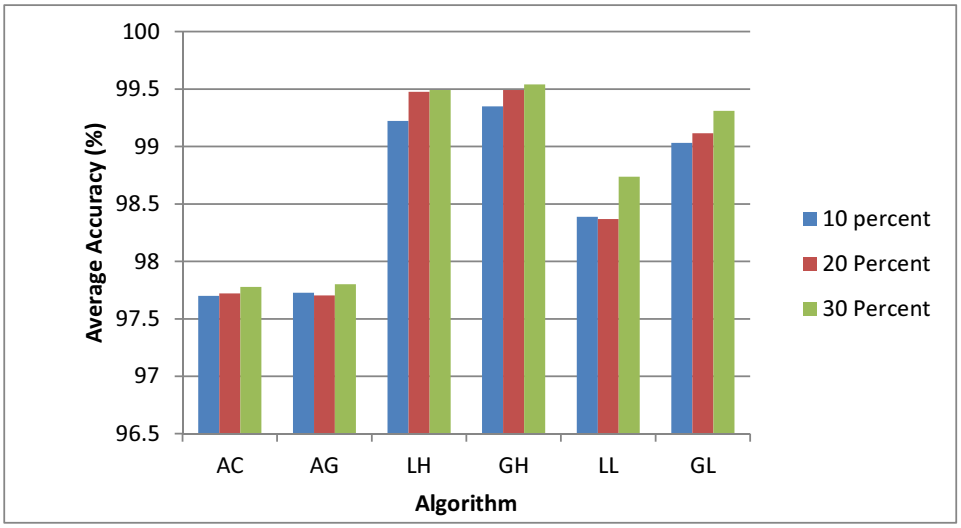

Figure 4.6: Average Accuracies for All Classifiers

While there is a statistically significant difference in the accuracies of the support vector machine classifiers, the practical differences between the SVM classifiers are small. Figures 4.7 and 4.8 present the average false positive and false negative rates for all of the classifiers. The CPU low-c SVM (LL) shows the lowest false positives, but the highest false negatives of the SVM classifiers. The high-c SVMs both perform similarly in the false negatives; however, the GPU (GH) implementation shows lower false positive rates. These results indicate that it is possible to take advantage of the GPU implementation's speedup with only a small impact on accuracy.

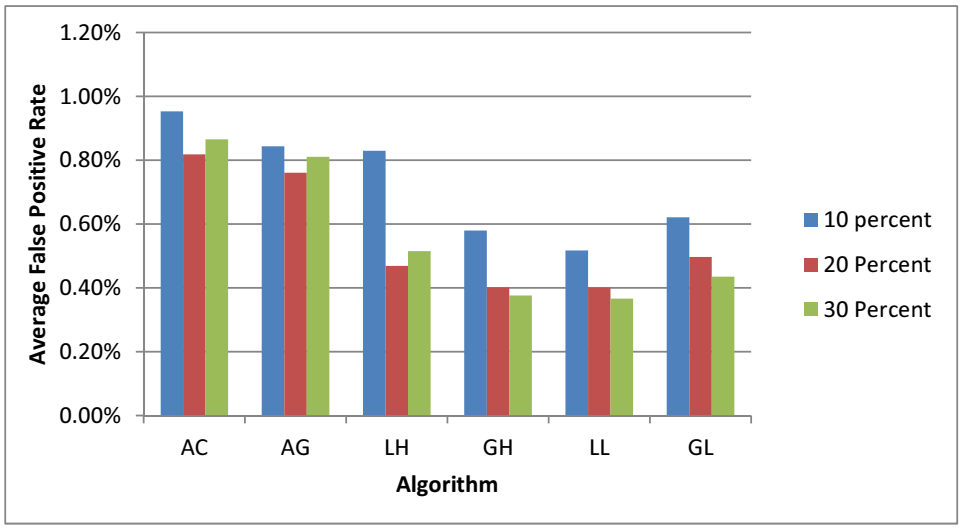

Figure 4.7: Average False Positive Rates

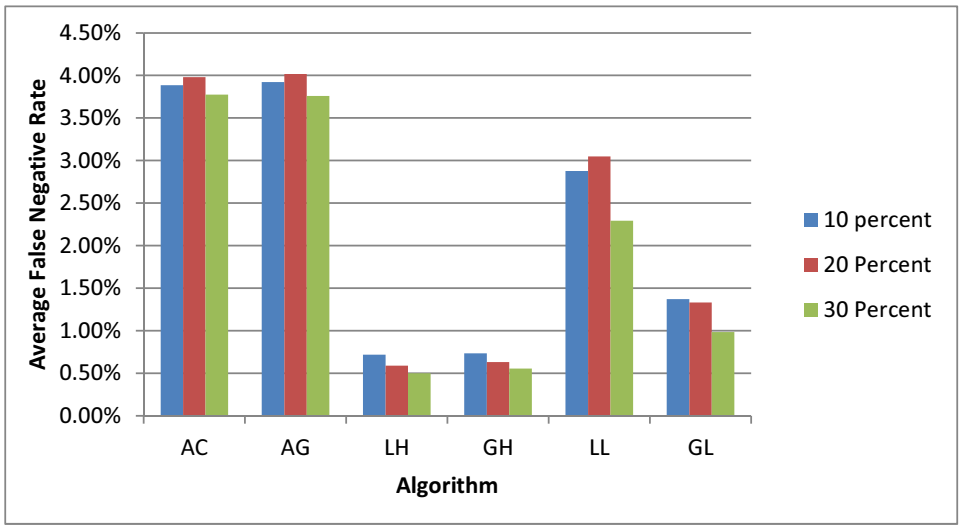

Figure 4.8: Average False Negative Rates

4.2 Results and Analysis of Experiment 2

In Experiment 2, the McPAD payload analysis technique is applied to a test dataset provided by Perdisci with the McPAD source code [Per09]. The 2_v-gram technique is used to create multiple datatsets, a 40 cluster set and a 160 cluster set, for seven different values

of ν. An ensemble of classifiers is created using varying numbers of base classifiers (BCs) and two ensemble combination methods. Each base classifier uses a different ν-value dataset. The relative execution times and final ensemble accuracies are compared for the different classifier types. This experiment compares the CPU (AC) and GPU (AG) implementations for ANN and a low cost parameter GPU SVM (GL) and a high cost parameter CPU SVM (LH) as those are the better performing support vector machines in Experiment 1 in terms of relative execution times. Section 4.2.1 examines the significance of the differences between the execution times caused by changing the dataset factor levels. Section 4.2.2 explores the ratios of execution times between the CPU and GPU implementations of the base classifiers. Section 4.2.3 compares the final accuracies of each of the ensemble configurations and discusses the significance of differences due to the factor level changes.

4.2.1 Impact of Datasets and Cluster Levels on Classifier Execution Times. The relative training times of the ensembles are compared using the execution times from the majority vote combination method factor level with seven base classifiers. These times represent all seven of the different datasets for each cluster level. The implementation for the final classifier combination method ensemble is comparable in the training stage, but the testing stage has an extra operation of converting the output into a format for use by the final classifier. As a result, these execution times are not used for the analysis of relative training and testing times.

As in Experiment 1, Tukey's Honest Significant Difference test is used to determine if the difference between mean execution times is significant for each machine learning algorithm at the different factor levels. A comparison of the training and testing times for all of the machine learning algorithms shows a significant difference between some of the datasets for each classifier. The specific datasets that differ vary from classifier to classifier. This holds for the 160 cluster datasets as well. Since the classifiers all respond

differently to the changes in datasets, each dataset is examined as a separate factor level. The R commands used are included in Appendix C.

An analysis of the differences in mean training and testing times for each classifier type when the number of clusters is changed is used to determine if the number of clusters makes a significant impact on the execution time of the base classifier. The results of the Tukey tests indicate that each classifier shows a significant difference between the cluster levels for every dataset for both training and testing. Since the change of cluster level is a four-fold difference in the number of input features per instance, it is expected to have a significant impact on the execution time of the classifiers. The R commands used are included in Appendix C.

4.2.2 Analysis of Differences in Execution Times. The differences between the average execution times for each of the classifier types provide an indication of the relative performance of each classifier on the McPAD processed datasets. R is used to compare the difference in means at the 90% confidence level between the base classifiers' train and test times. For every dataset at the 40 cluster factor level, all of the mean training times and the mean testing times are significantly different between the machine learning algorithms. The 160 cluster level shows similar results for all of the datasets. The R commands used are included in Appendix C.

4.2.2.1 Analysis of Differences in ANN Implementations. An examination of the mean training and testing times provides an indication of the degree of difference between the execution times of the CPU and GPU implementations of a classifier for the same dataset. The GPU ANN (AG) shows a consistent speedup over the CPU (AC) for training across the datasets as shown in Figures 4.9 and 4.10. The degree of speedup is substantially lower than in Experiment 1, however. This is attributable to the size of neural networks used in this experiment. Experiment 1 uses networks with a hidden layer of 32

neurons, whereas this experiment utilizes only 2 hidden neurons. The ANN speedups are not as large as the smaller network requires fewer calculations per training instance. The observed average speedup between the CPU and GPU ANN for the 40 cluster set is approximately 2-fold for the majority of the datasets. The 160 cluster set has more inputs to the neural network and shows a larger degree of speedup due to the additional calculations involved per training instance at an average increase of nearly 4-fold for the majority of the datasets.

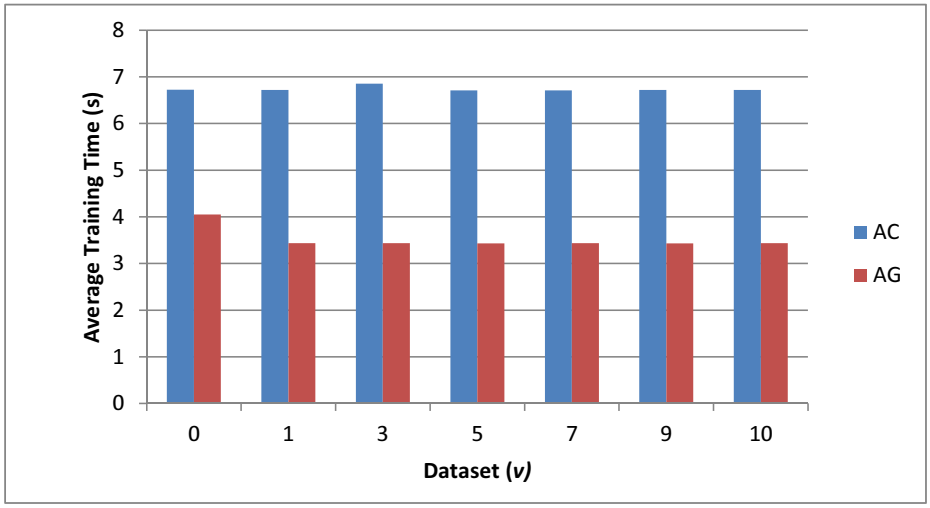

Figure 4.9: Mean Training Times of ANNs for 40 Cluster Datasets

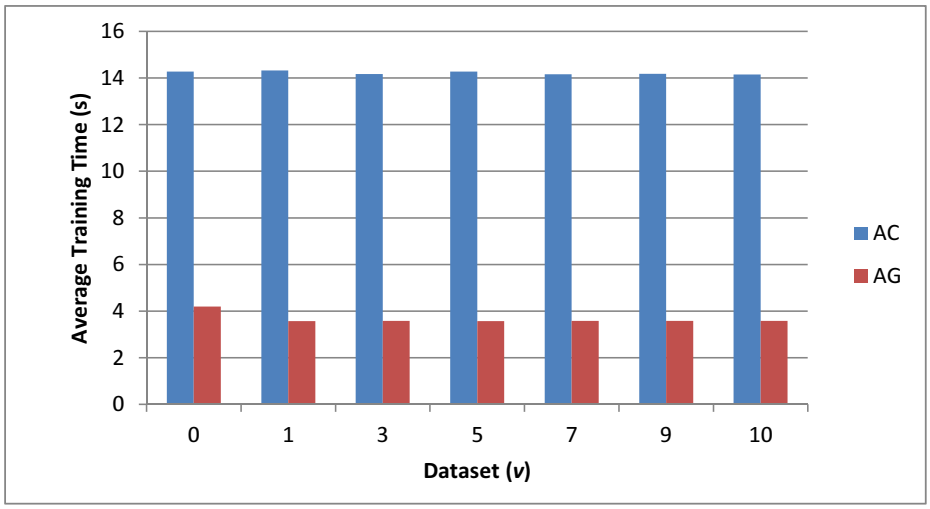

Figure 4.10: Mean Training Times of ANNs for 160 Cluster Datasets

The relative testing times for the ANN implementations show the opposite result of the training times. The CPU ANN is faster than the GPU ANN for all seven datasets. Again, this is attributable to the low number of hidden layer neurons limiting the number of calculations computable in parallel. As shown in Figures 4.11 and 4.12, the GPU performs only marginally better when the number of clusters is increased to 160. The negative impact of the low number of hidden layer neurons on the performance of the GPU ANN in the testing stage overwhelms any gain observed from increasing the number of input features.

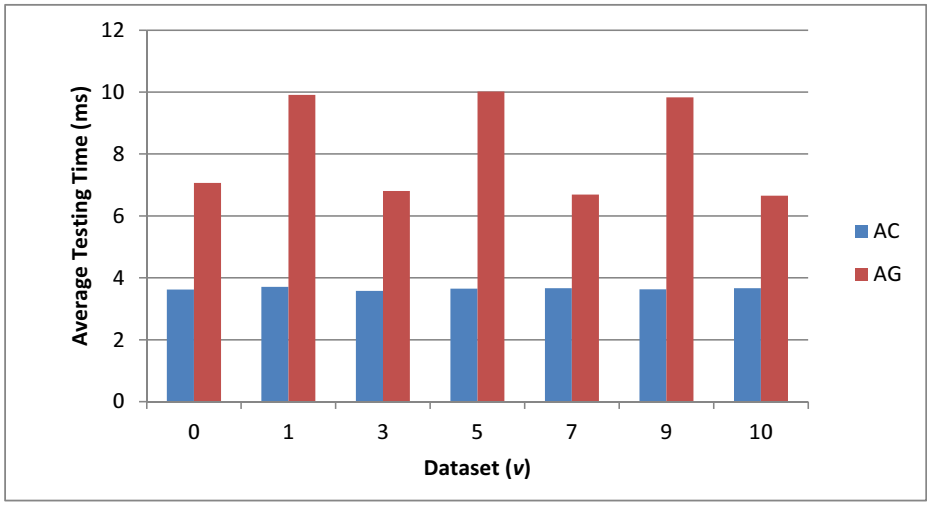

Figure 4.11: Mean Testing Times of ANNs for 40 Cluster Datasets

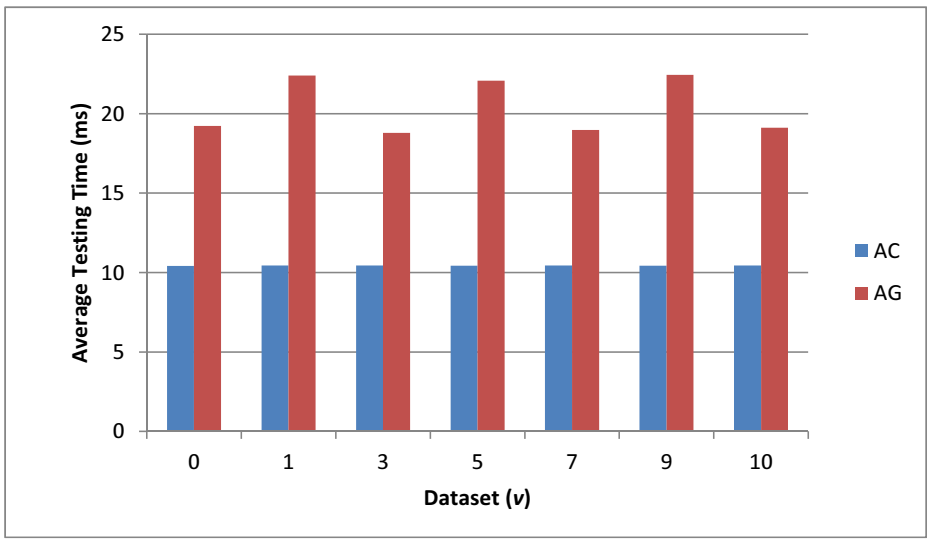

Figure 4.12: Mean Testing Times of ANNs for 160 Cluster Datasets

4.2.2.2 Analysis of Differences in SVM Implementations. The support vector machine classifiers show similar results as in Experiment 1. For all of the training datasets, the GPU (GL) is faster than the CPU (LH). The degree of speedup shows some variance due to the differences in the datasets as shown in Figures 4.13 and 4.14. For the 40 cluster datasets, the speedups are on the order of 2-5x; the 160 cluster datasets present much larger gains at 7x-15x. This is attributable to the impact of the number of input features on the different SVM implementations. The CPU shows an average increase in execution time of 4x when the clusters are increased, whereas the GPU shows an average increase of 1.4x as shown in Table 4.15.

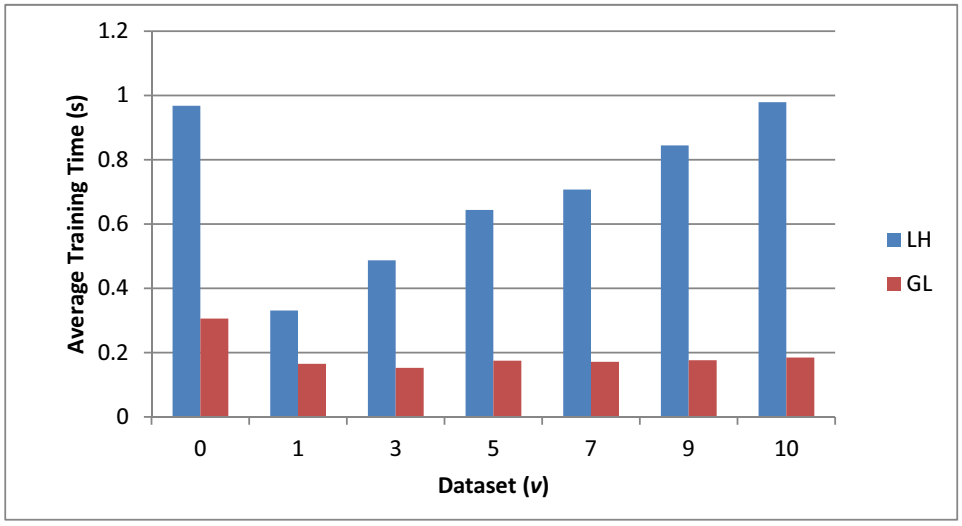

Figure 4.13: Mean Training Times of SVMs for 40 Cluster Datasets

Table 4.15: Impact Ratios of Cluster Changes for Average SVM Training Times

Alg	0	1	3	4	7	9	10	Mean	90% C.I.
LH	2.92	5.69	4.57	3.99	3.83	3.64	4.01	4.09	0.63
GL	1.31	1.43	1.44	1.37	1.46	1.45	1.40	1.41	0.04

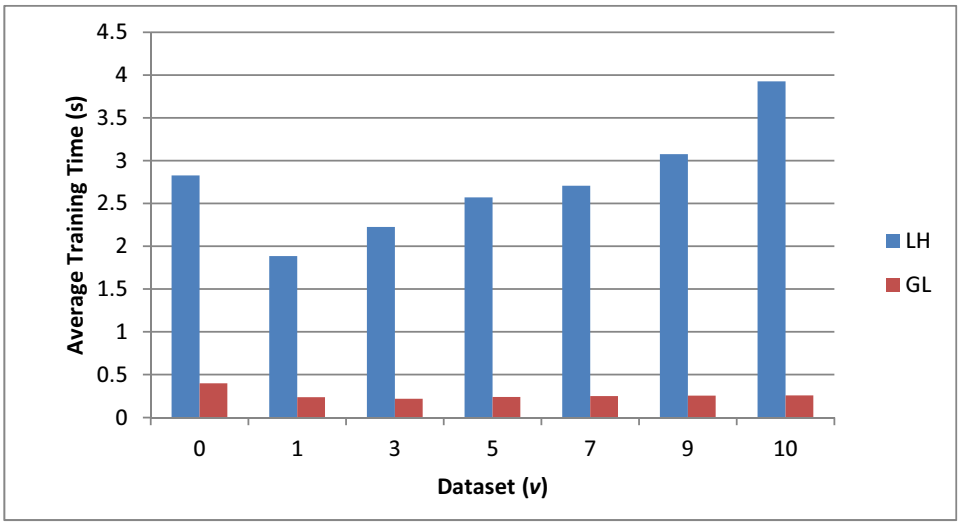

Figure 4.14: Mean Training Times of SVMs for 160 Cluster Datasets

The testing times for the support vector machine implementations at the 40 cluster level show marginal gains for the GPU implementation for four of the seven datasets; the others execute more quickly on the CPU implementation. At the higher cluster level, however, the GPU outperforms the CPU on all seven datasets. The mean testing times for the SVM implementations for 40 and 160 cluster levels are given in Figures 4.15 and 4.16, respectively. As with the training times, the increase in input features has a more significant impact on the CPU than the GPU. This is as expected since the GPU can increase its utilization when given more input features by launching more parallel thread blocks. The relative impact of the cluster size on the mean testing times is given in Table 4.16.

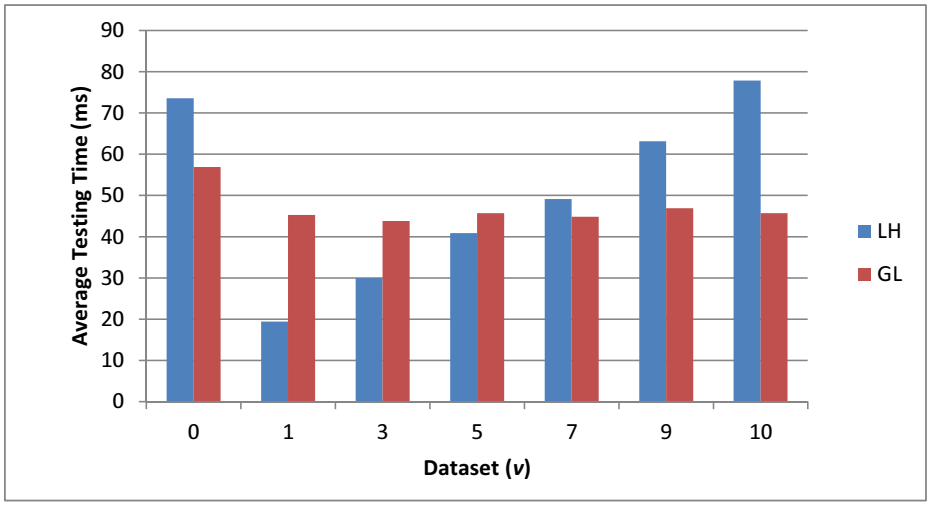

Figure 4.15: Mean Testing Times of SVMs for 40 Cluster Datasets

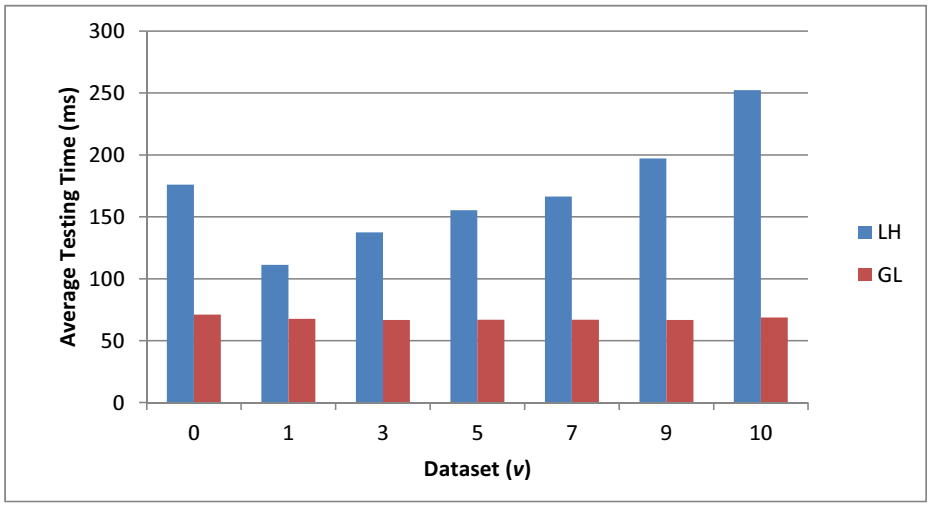

Figure 4.16: Mean Testing Times of SVMs for 160 Cluster Datasets

Table 4.16: Impact Ratios of Cluster Changes for Average SVM Testing Times

Alg	0	1	3	4	7	9	10	Mean	90% C.I.
CPU	2.39	5.72	4.59	3.80	3.39	3.12	3.24	3.75	0.81
GPU	1.25	1.49	1.52	1.46	1.49	1.43	1.50	1.45	0.07

Overall, as in Experiment 1, the GPU implementations show a performance gain over the CPU implementations in the training phase for all datasets. The testing phase, however, indicates that the low number of hidden neurons severely limits the potential performance gains of the GPU ANN over the CPU ANN. The GPU SVMs performance gain is limited by the number of input features as indicated by the difference in scaling between the CPU and GPU implementations when the input features are increased. These results indicate that the GPU SVM is best utilized when a larger number of input features is used.

4.2.3 Analysis of Ensemble Accuracy. This section evaluates the overall prediction accuracies for the machine learning ensemble classifiers on the McPAD workload to determine the most effective ensemble configuration among those considered. The overall accuracy for each ensemble configuration is compared. The differences provide an indication of the effects of the number of clusters, the number of base classifiers, and the combination method on the overall prediction accuracy of the ensemble. As with Experiment 1, a non-parametric test is used to determine if there is a significant difference between the factor levels at the 0.1 significance level. The ensemble accuracies are evaluated using Wilcoxson's signed-rank test in R. Section 4.2.3.1 compares the accuracies of the ensembles for each of the different machine learning algorithms. The impact of changing the number of base classifiers used in the ensemble is examined in Section 4.2.3.2. Section 4.2.3.3 discusses the effects of the ensemble combination method

on the overall accuracy. Lastly, Section 4.2.3.4 analyzes the differences in accuracies between the different cluster factor levels.

4.2.3.1 Impact of Machine Learning Algorithm on Accuracy. Due to the nature of support vector machine classifiers, it is expected that the accuracy for the voting ensembles be constant across the replications. The final classifier ensembles add the variation of a randomly initialized artificial neural network to the final determination, resulting in the potential for variation between replications. The CPU SVM ensembles behave as expected. The voting ensembles show the same accuracy across all replications for both cluster levels, while the classifier method shows variation. The GPU SVM, however, does not behave as expected. An examination of the accuracies for the GPU SVM ensembles shows an inconsistency in accuracies between the replications for all of the 40 cluster and some of the 160 cluster factor levels. As this is not encountered in Experiment 1, it is assumed that there is a rounding error in the implementation of the GPU SVM, causing examples to be classified differently from one run to the next for specific datasets. A full examination of this issue is not within the scope of this research and is left to future work. The accuracies for the GPU SVM voting ensembles are provided in Appendix D.

An analysis of the differences in accuracies between the CPU and GPU implementations of each machine learning algorithm shows similar results as Experiment 1. The Wilcoxon tests indicate that there is no significant difference between the CPU (AC) and GPU (AG) artificial neural network implementations for any of the factor levels. The CPU SVM (LH) ensembles show a significant difference from all of the other algorithms for each factor level. The GPU SVM (GL), however, shows no difference between it and the ANN ensembles at several factor levels. The factor levels that show no difference between the GPU SVM and CPU ANN also show no difference between the GPU SVM and GPU ANN for all of the three and five BC configurations. The seven BC

configuration, however, only matches for half of the factor levels that show no difference between the GPU SVM and the CPU ANN. From the low p-values, it is possible that there is a significant difference between these classifiers that is undetected due to the low power of the test with this number of replications. The R commands used to conduct the Wilcoxon tests are included in Appendix C.

4.2.3.2 Impact of Adding Base Classifiers on Accuracy. An examination of the ensemble accuracies shows that there is a significant impact for most of the factor levels of the 40 cluster ensemble when the number of base classifiers is changed. The three BC and five BC ensembles with the majority vote combination method show no significant difference in accuracy at the 0.10 significance level for the ANN ensembles. Likewise, the five BC and seven BC ensembles using the final classifier method show no significant difference in accuracy for any of the algorithms. The rest of the configurations do show a significant difference at the 0.10 level. Similarly, the 160 cluster ensembles also show a significant difference for all of the factor levels except the five and seven BC final classifier levels. The R commands used to conduct the Wilcoxon tests are included in Appendix C.

At the 40 cluster level, the addition of base classifiers from three to five does not significantly change the accuracy for the ANN voting ensemble and the increase to seven base classifiers decreases the accuracy. As shown by the average accuracies of the voting ensembles in Figure 4.17, the CPU support vector machine ensembles show a minimal decrease between the three and five levels and an increase for the seven classifier level. Like the ANN ensemble, the GPU SVM ensemble records a decrease for the seven base classifier level.

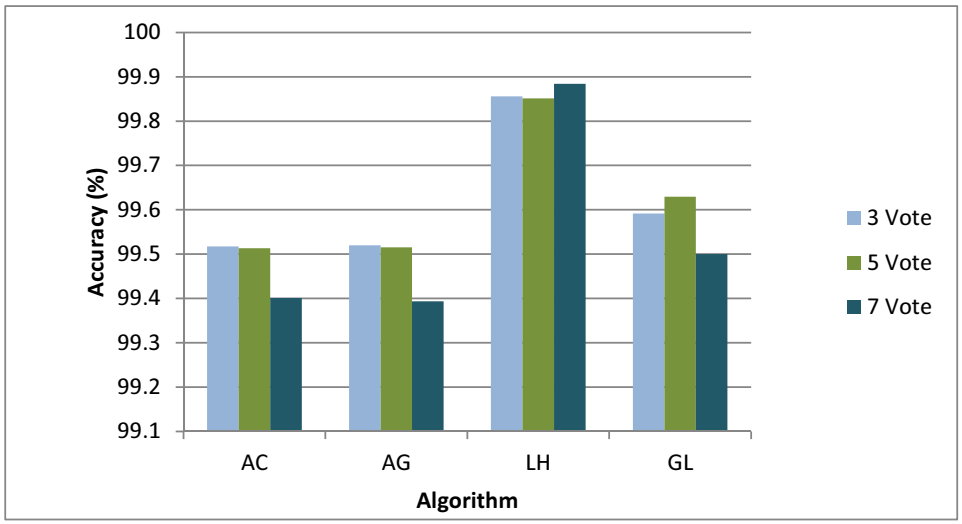

Figure 4.17: Average Accuracies for 40 Cluster Voting Ensembles

The addition of base classifiers to an ensemble increases the amount of information used to make the final classification of the traffic sample. As each base classifier uses a different ν dataset value, some will reach different determinations than others for the same traffic sample. As more incorrect base classifier votes are added, the correct votes are overruled, resulting in a decrease in accuracy. An examination of the base classifier accuracies indicates that the ANN and GPU SVM accuracies are lower for the two additional datasets in the seven BC configuration than the CPU SVM, resulting in a decrease in accuracy rather than an increase at that factor level.

All of the ensembles demonstrate higher accuracies for the final classifier combination method at the five and seven BC levels as compared to the three classifier level, but with no significant difference between the five and seven levels as indicated by the Wilcoxon tests. The average accuracies for each final classifier ensemble at the 40 cluster level are shown in Figure 4.18. The results for the 160 cluster ensembles are similar and are included in Appendix D.

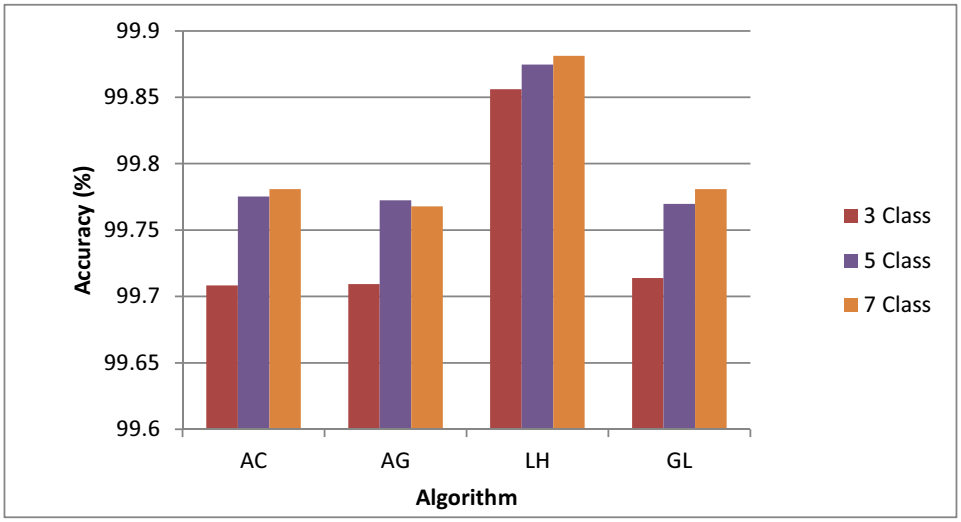

Figure 4.18: Average Accuracies for 40 Cluster Final Classifier Ensembles

4.2.3.3 Impact of Ensemble Combination Method on Accuracy. The accuracies for the majority vote ensembles and the final classifier ensembles are compared to determine the impact of the combination method on the final accuracy. For the 40 cluster datasets, the Wilcoxon signed-rank test indicates that there is a significant difference between the voting ensemble and the final classifier ensemble for almost all of the base classifier types. The CPU support vector machine reports the same accuracies for the voting ensemble and final classifier ensemble at the three BC level for each replication. As a result, the comparison is not conducted with R; it is treated as if there is no significant difference since all of the differences in accuracy are zero. Additionally, the CPU support vector machine BC ensemble shows no significant difference at the seven BC level.

At the 160 cluster level, the Wilcoxon test indicates that there is a significant difference between the voting ensemble and the final classifier ensemble combination methods for all of the classifier configurations at the seven base classifier level. The five BC level shows no significant difference for the the CPU SVM and the three BC level shows no significant difference for the GPU ANN and both of the SVMs. Due to the low

p-values, it is possible that there is a significant difference at these factor levels that is undetected due to the low power of the test with this number of replications.

In general, the observed accuracies are higher for the classifier ensembles than the voting ensembles. This increase in accuracy is due to the added flexibility of the neural network classifier method over the majority voting method. Since the classifier determines the weights of each base classifier's inputs based on its accuracy in the training phase, it can emphasize the more accurate base classifiers over the less accurate ones. In contrast, the majority vote ensemble gives all base classifiers an equal weight in the final output regardless of their historical accuracy. Each of the classifier configurations that shows a statistically significant difference also shows an increase in accuracy when the combination method is changed from majority vote to final classifier as illustrated in Figure 4.19.

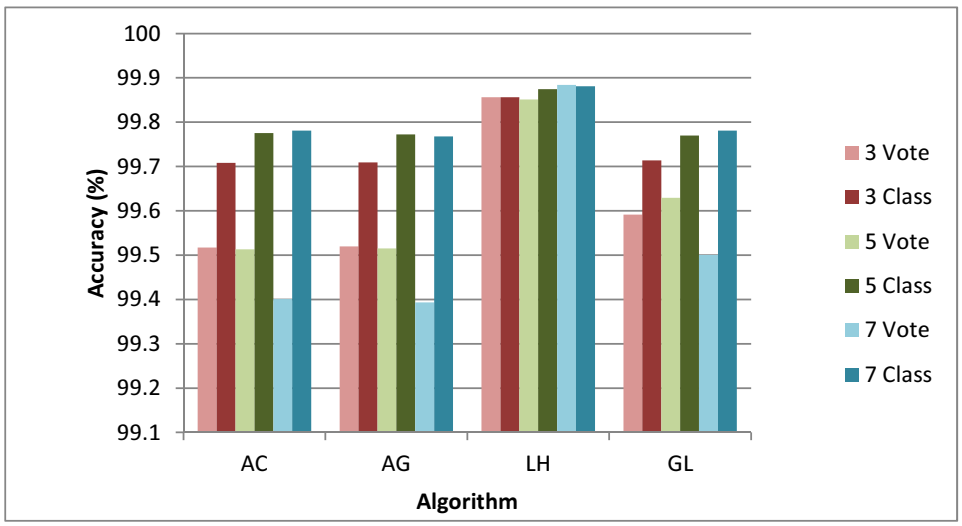

Figure 4.19: Average Accuracies for All 40 Cluster Ensembles

4.2.3.4 Impact of Cluster Size on Accuracy. The significance tests for the differences between the accuracies show a significant difference between the 40 cluster

and 160 cluster ensembles for the all of the factor levels. The differences between the cluster levels is expected as the payload processing and clustering method employed results in a different representation of the underlying data for different numbers of clusters. For the majority of the configurations, the 40 cluster ensembles perform better than their 160 cluster equivalents. There is one exception, however. The CPU support vector machine performs marginally better with the 160 cluster datasets than the 40 cluster datasets for one factor level, the five base classifier majority voting ensemble. It shows an average accuracy of 98.85% for the 40 cluster dataset and 98.86% for the 160 cluster dataset. This indicates that better accuracy results may be attained by using different combinations of cluster levels and algorithms in the same ensemble and is left to future research.

Overall, the best performance for the 40 cluster dataset is observed at the five base classifier level with the final classifier combination method. At this factor level, the GPU SVM and ANN implementations are not significantly different as discussed in Section 4.2.3.1. In terms of overall accuracy, the CPU SVM performs the best; however, as in Experiment 1, the practical differences in accuracy are small. Figure 4.20 presents the false positive and false negative rates for the five base classifier ensembles with the final classifier combination method. The CPU support vector machine (LH) ensemble shows the lowest false positive rate, but a higher false negative rate than the GPU SVM (GL) ensemble. The GPU SVM shows the highest false positive rate, but the lowest false negative rate. The false positive and false negative rates for the remaining factor levels are included in Appendix D.

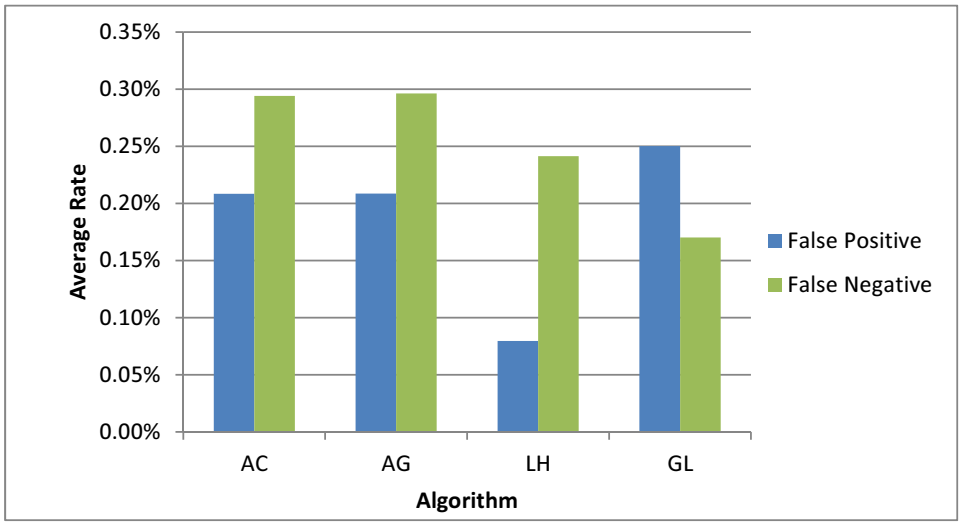

Figure 4.20: False Positive and Negative Rates for 5 BC Final Classifier 40 Cluster Ensembles

4.3 Analysis Highlights

The results from Experiments 1 and 2 provide an indication of the differences in execution time performance for machine learning-based intrusion detection systems when executed on the CPU and GPU. Experiment 1 shows that the GPU implementation is less affected by increases in dataset size than the CPU implementation. It also provides an indication of the impact of changing the support vector machine cost and gamma parameters on the relative execution times of each implementation. The performance of the CPU support vector machines indicate that LIBSVM performs 1.9-3.2x better with a high cost parameter for the NSL-KDD dataset, while GTSVM performs 8-13x better in the training stage with a lower cost parameter. The accuracies of the classifiers in Experiment 1 indicate that the support vector machines are more accurate at classifying the NSL-KDD dataset than the artificial neural networks, with accuracies near 99% and 97.5%, respectively. The low differences in accuracies between the CPU high-c SVM and GPU low-c SVM indicate that it is possible to take advantage of the speed differences afforded by the GPU with only a small impact on accuracy.

Experiment 2 demonstrates the effectiveness of utilizing supervised machine learning techniques on network payload data processed using the McPAD 2_v-gram technique. The results of the execution time comparisons indicate that the performance of the GPU ANN implementation is more highly dependent on the number of hidden neurons in the neural network than the number of input features in the dataset. The GPU support vector machine, however, shows an increase in relative performance when the number of input features is increased. Unlike Experiment 1, the ANN accuracy performance is not significantly different from the GPU SVM at the most accurate factor level; however, it is still significantly slower in execution time. These results demonstrate that it is possible to obtain comparable accuracies for 2_v-gram payload data with a 2-15x increase in performance using the GPU machine learning libraries.

4.4 Summary

This chapter presents and analyzes the results from two experiments. A statistical analysis of the differences in execution times and accuracies is performed for both experiments. Finally, an overall analysis of the results is provided. The results show that the GPU support vector machine implementation provides a 10-85x speedup over the CPU implementation for the NSL-KDD dataset and a 2-15x speedup for the McPAD dataset. The GPU artificial neural network implementation provides speedups of up to 29x over the CPU ANN implementation. These results demonstrate that it is possible to take advantage of the speedup provided by the GPU when classifying network traffic.

5 Conclusions

This chapter presents the conclusions drawn from the research. Section 5.1 compares the results of the experiments with the research goals to determine if the objectives were met. The impact of this research is presented in Section 5.2. Lastly, Section 5.3 presents recommendations for future research.

The hypothesis of the first research goal is that support vector machines will be more accurate at classifying anomalous network traffic using the GPU than the artificial neural networks. The second goal hypothesizes that the GPU implementation for the intrusion detection system will be significantly faster than the CPU implementation. For goal three, the hypothesis is that adding more base classifiers to the ensemble and using a final classifier combination method instead of a simple majority vote will increase the accuracy of the ensemble when classifying network payload data processed using the 2_ν-gram technique from McPAD [PAF09].

5.1 Conclusions of Research

5.1.1 Goal 1: Determine the Relative Accuracy of the GPU Machine Learning Anomaly Detection Algorithms. The first goal of this research is to determine whether using Gaussian kernel support vector machines (SVMs) or back-propagation learning artificial neural networks (ANNs) is the most effective method for classifying anomalous network traffic on the GPU. The results of Experiment 1 indicate that the support vector machines are more accurate for the NSL-KDD preprocessed traffic data. The results of Experiment 2 indicate that the most effective GPU ANN and GPU SVM ensemble configurations are not significantly different in accuracy.

5.1.2 Goal 2: Measure the Relative Performances of the CPU and GPU Implementations of the Anomaly Detector. The second goal of this research is to evaluate

the relative performance of the GPU anomaly detection method against the CPU implementation for the same datasets. GNIDS is evaluated using the NSL-KDD datasets and the McPAD processed datasets. The results of Experiment 1 indicate that the GPU implementation is significantly less affected by increases in dataset size than the CPU implementation. Additionally, the GPU implementation is faster for both training and testing on the NSL-KDD workloads. Experiment 2 shows that the performance gain of the GPU ANN over the CPU ANN is highly dependent on the number of hidden neurons used in the neural network. A performance gain is observed for the training phases, but the testing phase is more severely impacted by the lack of hidden neurons, resulting in faster classification using the CPU ANN. The training phases for the SVM classifier are faster on GPU than the CPU for both cluster levels. The classification phase, however, only shows a performance gain for the 160 cluster datasets, indicating the CPU SVM is more heavily impacted by the change in input features than the GPU SVM.

5.1.3 Goal 3: Determine the Relative Accuracy of Varying Ensemble Configurations.

The third goal of this research is to explore the effects of changing the number of base classifiers and ensemble combination method on the overall accuracy of a machine learning ensemble. GNIDS is used in an ensemble configuration with each of the machine learning implementations and is tested with multiple ensemble configurations on 2_v-gram processed datasets. The results indicate that there is only a marginal difference in accuracy between the best performing GPU ANN and GPU SVM configurations. The overall best performing configuration is the five base classifier ensemble using the 40 cluster datasets. Of the machine learning algorithms, the CPU SVM performed the best at this level; however, the GPU SVM shows comparable accuracies with only marginally higher false positive rates and lower false negative rates.

5.2 Impact of Research

This research explores the performance characteristics of network intrusion detection using supervised machine learning on the GPU. It also demonstrates the feasibility of using an ensemble of supervised machine learning classifiers to classify network traffic data processed using the 2_v-gram feature clustering technique. It is shown that both GPU artificial neural networks and GPU support vector machines can be effectively used to classify network payload data. This research provides an implementation of a flexible classification tool for GPU-accelerated supervised machine learning ensembles. As a result of using the GPU, GNIDS is less impacted by large datasets than a CPU-based implementation and can be deployed to any machine with a compatible GPU or can be executed using the included CPU implementation. It is easily expandable to support other machine learning algorithms and ensemble combination methods.

5.3 Recommendations for Future Work

The following are recommended areas for future work and expansion of GNIDS:

- Migrate the machine learning algorithm implementations to OpenCL or AMP C++ to take advantage of hardware flexibility. OpenCL provides a parallel coding standard similar to CUDA that has implementations for multi-core CPUs and both NVIDIA and AMD graphics hardware [Khr12]. AMP C++ is a new API announced by Microsoft that takes advantage of the GPU's parallelism at the driver level, allowing compatibility across hardware vendors [Mic12].

- Integrate the 2_v-gram payload processing technique so that the system can process its own packets for analysis rather than relying on a preprocessed dataset.

- Implement unsupervised machine learning techniques to provide the ability for the detector method to be trained using unlabeled traffic. It is suggested that one-class

support vector machines be implemented along with self-organizing maps or some other clustering technique.

- Implement alert generation and logging functionality.
- Evaluate performance impact of deploying GNIDS to monitor traffic at the individual network host level.
- Evaluate the impact of using a low-end GPU versus a high-end GPU on system performance.

5.4 Summary

This research focuses on the implementation and evaluation of a graphics processing unit accelerated network intrusion detection system (GNIDS) that uses the parallel nature of the GPU to perform network anomaly detection using supervised machine learning techniques. GNIDS is designed to support multiple machine learning classifiers in an ensemble setup using two different ensemble combination methods, a majority vote and a neural network classifier. Overall, results of this research indicate that it is possible to take advantage of the performance increases afforded by the GPU in the area of supervised machine learning intrusion detection.

Appendix A: Supplemental Data for Experiment 1

A.1 R Script - Tukey Test for Experiment 1

```
#data entry
times_KDD = read.csv("Z:\\Fall\ 11\\chapter\ 4\\kddlogs\\Times.csv",header=TRUE)
times_KDD_10_41 = times_KDD[grep("10",times_KDD$Size),]
times_KDD_20_41 = times_KDD[grep("20",times_KDD$Size),]
times_KDD_30_41 = times_KDD[grep("30",times_KDD$Size),]
times_KDD_41 = times_KDD[grep("41",times_KDD$Features),]
#difference of Alg at each Size for trainining
y_train_KDD_10_41 = TukeyHSD(aov(lm(Train~factor(Alg),times_KDD_10_41)),conf.level = 0.90)
y_train_KDD_10_41
y_train_KDD_20_41 = TukeyHSD(aov(lm(Train~factor(Alg),times_KDD_20_41)),conf.level = 0.90)
y_train_KDD_20_41
y_train_KDD_30_41 = TukeyHSD(aov(lm(Train~factor(Alg),times_KDD_30_41)),conf.level = 0.90)
y_train_KDD_30_41
#difference of Alg at each Size for testing
y_test_KDD_10_41 = TukeyHSD(aov(lm(Test~factor(Alg),times_KDD_10_41)),conf.level = 0.90)
y_test_KDD_10_41
y_test_KDD_20_41 = TukeyHSD(aov(lm(Test~factor(Alg),times_KDD_20_41)),conf.level = 0.90)
y_test_KDD_20_41
y_test_KDD_30_41 = TukeyHSD(aov(lm(Test~factor(Alg),times_KDD_30_41)),conf.level = 0.90)
y_test_KDD_30_41
#impact on Alg of Size for train
#impact on AC
y_train_KDD_41_AC = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("AC",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_AC
#impact on AG
y_train_KDD_41_AG = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("AG",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_AG
#impact of GH
y_train_KDD_41_G1 = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("G1",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_G1
#impact on LH
y_train_KDD_41_L1 = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("L1",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_L1
#impact on GL
y_train_KDD_41_G2 = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("G2",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_G2
#impact on LL
y_train_KDD_41_L2 = TukeyHSD(aov(lm(Train~factor(Size),times_KDD_41[grep("L2",times_KDD_41$Alg),])),conf.level = 0.90)
y_train_KDD_41_L2
#impact on Alg of Size for test
#impact on AC
y_test_KDD_41_AC = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("AC",times_KDD_41$Alg),])),conf.level = 0.90)
y_test_KDD_41_AC
#impact on AG
y_test_KDD_41_AG = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("AG",times_KDD_41$Alg),])),conf.level = 0.90)
y_test_KDD_41_AG
#impact of GH
y_test_KDD_41_G1 = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("G1",times_KDD_41$Alg),])),conf.level = 0.90)
```

```
y_test_KDD_41_G1
#impact on LH
y_test_KDD_41_L1 = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("L1",times_KDD_41$Alg),])),conf.level = 0.90)
y_test_KDD_41_L1
#impact on GL
y_test_KDD_41_G2 = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("G2",times_KDD_41$Alg),])),conf.level = 0.90)
y_test_KDD_41_G2
#impact on LL
y_test_KDD_41_L2 = TukeyHSD(aov(lm(Test~factor(Size),times_KDD_41[grep("L2",times_KDD_41$Alg),])),conf.level = 0.90)
y_test_KDD_41_L2
```

A.2 R Script - Wilcoxon Test Experiment 1

```
library(coin)
scores_KDD = read.csv("Z:\\Fall\ 11\\chapter\ 4\\kddlogs\\Scores.csv",header=TRUE)
ac_KDD_df= scores_KDD[grep("AC",scores_KDD$Alg),]
ag_KDD_df= scores_KDD[grep("AG",scores_KDD$Alg),]
g1_KDD_df= scores_KDD[grep("G1",scores_KDD$Alg),]
l1_KDD_df= scores_KDD[grep("L1",scores_KDD$Alg),]
g2_KDD_df= scores_KDD[grep("G2",scores_KDD$Alg),]
l2_KDD_df= scores_KDD[grep("L2",scores_KDD$Alg),]
#Group by Alg and Size Level.
ac_10 = c(ac_KDD_df[grep("10",ac_KDD_df$Size),]$Score)
ac_20 = c(ac_KDD_df[grep("20",ac_KDD_df$Size),]$Score)
ac_30 = c(ac_KDD_df[grep("30",ac_KDD_df$Size),]$Score)
ag_10 = c(ag_KDD_df[grep("10",ag_KDD_df$Size),]$Score)
ag_20 = c(ag_KDD_df[grep("20",ag_KDD_df$Size),]$Score)
ag_30 = c(ag_KDD_df[grep("30",ag_KDD_df$Size),]$Score)
g1_10 = c(g1_KDD_df[grep("10",g1_KDD_df$Size),]$Score)
g1_20 = c(g1_KDD_df[grep("20",g1_KDD_df$Size),]$Score)
g1_30 = c(g1_KDD_df[grep("30",g1_KDD_df$Size),]$Score)
l1_10 = c(l1_KDD_df[grep("10",l1_KDD_df$Size),]$Score)
l1_20 = c(l1_KDD_df[grep("20",l1_KDD_df$Size),]$Score)
l1_30 = c(l1_KDD_df[grep("30",l1_KDD_df$Size),]$Score)
g2_10 = c(g2_KDD_df[grep("10",g2_KDD_df$Size),]$Score)
g2_20 = c(g2_KDD_df[grep("20",g2_KDD_df$Size),]$Score)
g2_30 = c(g2_KDD_df[grep("30",g2_KDD_df$Size),]$Score)
l2_10 = c(l2_KDD_df[grep("10",l2_KDD_df$Size),]$Score)
l2_20 = c(l2_KDD_df[grep("20",l2_KDD_df$Size),]$Score)
l2_30 = c(l2_KDD_df[grep("30",l2_KDD_df$Size),]$Score)
#----------difference in Algorithm by Size----------------
#diff at 10
p_ALG_10=c(
pvalue(wilcoxsign_test(ag_10~ac_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_10~ac_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_10~ac_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_10~ac_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_10~ac_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_10~ag_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_10~ag_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_10~ag_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_10~ag_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_10~g1_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_10~g1_10,distribution="exact",zero.method="Pratt")),
```

```
pvalue(wilcoxsign_test(l2_10~g1_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_10~l1_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_10~l1_10,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_10~g2_10,distribution="exact",zero.method="Pratt")))
write(p_ALG_10,"",1)
#diff at 20
p_ALG_20=c(
pvalue(wilcoxsign_test(ag_20~ac_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_20~ac_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_20~ac_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_20~ac_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_20~ac_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_20~ag_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_20~ag_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_20~ag_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_20~ag_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_20~g1_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_20~g1_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_20~g1_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_20~l1_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_20~l1_20,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_20~g2_20,distribution="exact",zero.method="Pratt")))
write(p_ALG_20,"",1)
#diff at 30
p_ALG_30=c(
pvalue(wilcoxsign_test(ag_30~ac_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_30~ac_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_30~ac_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_30~ac_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_30~ac_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g1_30~ag_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_30~ag_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_30~ag_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_30~ag_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l1_30~g1_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_30~g1_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_30~g1_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(g2_30~l1_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_30~l1_30,distribution="exact",zero.method="Pratt")),
pvalue(wilcoxsign_test(l2_30~g2_30,distribution="exact",zero.method="Pratt")))
write(p_ALG_30,"",1)
```

A.3 Workload Characteristics

Table A.1: Traffic Distribution for NSL-KDD 20% Subset

Traffic Class	Samples	% Total Samples
Normal Traffic	13468	53.459%
DoS Traffic	9185	36.459%
User to Root Traffic	10	0.040%
Remote to Local Traffic	199	0.790%
Probe Traffic	2331	9.253%

Table A.2: Traffic Distribution for NSL-KDD 30% Subset

Traffic Class	Samples	% Total Samples
Normal Traffic	20202	53.460%
DoS Traffic	13778	36.460%
User to Root Traffic	15	0.040%
Remote to Local Traffic	298	0.789%
Probe Traffic	3496	9.251%

Appendix B: Experimental Data for Experiment 1

B.1 Training Times

Table B.1: Training Times (s) for CPU ANN

Run	10	20	30
1	260.9045	524.4233	800.6278
2	261.3478	526.2334	799.7555
3	260.6942	525.1440	800.2352
4	260.9887	525.3850	800.2097
5	260.1812	525.5359	799.4652

Table B.2: Training Times (s) for GPU ANN

Run	10	20	30
1	9.4720	18.4313	27.4207
2	9.4559	18.4283	27.4257
3	9.4661	18.4274	27.4179
4	9.4648	18.4296	27.4277
5	9.4701	18.4299	27.4268

Table B.3: Training Times (s) for GPU SVM-High

Run	10	20	30
1	4.2367	7.5584	15.4567
2	4.3578	7.8402	14.9039
3	4.5921	7.5567	15.3311
4	4.1744	7.5516	15.7191
5	4.2792	7.8409	15.0665

Table B.4: Training Times (s) for CPU SVM-High

Run	10	20	30
1	5.2349	27.9568	95.0705
2	5.2200	28.1045	95.4993
3	5.2481	28.3211	95.5483
4	5.2629	28.1644	96.4790
5	5.2304	28.3090	96.5413

Table B.5: Training Times (s) for GPU SVM-Low

Run	10	20	30
1	0.4988	0.7102	1.1123
2	0.4850	0.7115	1.1737
3	0.4864	0.7339	1.0890
4	0.4977	0.7146	1.1063
5	0.4831	0.7525	1.1429

Table B.6: Training Times (s) for CPU SVM-Low

Run	10	20	30
1	11.9461	90.6718	187.1990
2	11.9984	90.6824	187.9231
3	12.0361	90.0523	190.2981
4	12.0703	90.0338	187.0447
5	12.0157	90.2141	187.9421

B.2 Testing Times

Table B.7: Testing Times (ms) for CPU ANN

Run	10	20	30
1	19.1009	38.0686	57.0920
2	19.1978	38.2302	57.0863
3	19.1159	38.2076	56.7110
4	19.1040	38.0241	56.8088
5	19.1337	37.8701	56.8083

Table B.8: Testing Times (ms) for GPU ANN

Run	10	20	30
1	7.7418	10.2129	12.6419
2	7.7787	10.2521	12.5407
3	7.7813	10.2209	12.6778
4	7.6795	10.4127	12.5108
5	7.8517	10.1542	12.4317

Table B.9: Testing Times (ms) for GPU SVM-High

Run	10	20	30
1	41.5133	72.6751	99.4987
2	41.7179	71.2621	99.4714
3	44.0194	73.0409	100.6619
4	40.7030	71.8097	101.2230
5	41.1058	72.8212	100.9495

Table B.10: Testing Times (ms) for CPU SVM-High

Run	10	20	30
1	336.9118	957.5062	1790.9575
2	336.4334	953.7136	1803.7802
3	335.8057	967.9052	1801.4094
4	335.9800	953.3447	1810.2800
5	335.8952	959.2463	1803.1212

Table B.11: Testing Times (ms) for GPU SVM-Low

Run	10	20	30
1	43.3857	74.0620	106.2155
2	43.5041	74.5227	105.5904
3	43.2079	74.4842	103.4442
4	44.0608	73.6946	106.4497
5	42.7844	74.8246	105.4607

Table B.12: Testing Times (ms) for CPU SVM-Low

Run	10	20	30
1	649.3035	2218.0776	4895.9872
2	649.3771	2231.8417	4868.2795
3	648.6016	2212.7901	4892.4358
4	648.7443	2221.0593	4856.2410
5	649.2062	2211.6358	4835.3543

B.3 Accuracies

Table B.13: Accuracies for CPU ANN

CPU ANN	10	20	30
1	97.682	97.730	97.796
2	97.682	97.741	97.761
3	97.666	97.757	97.812
4	97.650	97.686	97.716
5	97.825	97.690	97.804

Table B.14: Accuracies for GPU ANN

GPU ANN	10	20	30
1	97.634	97.741	97.777
2	97.690	97.678	97.806
3	97.713	97.797	97.790
4	97.809	97.698	97.785
5	97.793	97.606	97.846

Table B.15: Accuracies for GPU SVM-H

GPU SVM-H	10	20	30
1	99.349	99.492	99.540
2	99.349	99.492	99.540
3	99.349	99.492	99.540
4	99.349	99.492	99.540
5	99.349	99.492	99.540

Table B.16: Accuracies for CPU SVM-H

CPU SVM-H	10	20	30
1	99.222	99.476	99.492
2	99.222	99.476	99.492
3	99.222	99.476	99.492
4	99.222	99.476	99.492
5	99.222	99.476	99.492

Table B.17: Accuracies for GPU SVM-L

GPU SVM-L	10	20	30
1	99.031	99.115	99.309
2	99.031	99.115	99.309
3	99.031	99.115	99.309
4	99.031	99.115	99.309
5	99.031	99.115	99.309

Table B.18: Accuracies for CPU SVM-L

CPU SVM-L	10	20	30
1	98.388	98.369	98.738
2	98.388	98.369	98.738
3	98.388	98.369	98.738
4	98.388	98.369	98.738
5	98.388	98.369	98.738

B.4 False Positive Rates

Table B.19: False Positive Rates for CPU ANN

CPU ANN	10	20	30
1	0.83%	0.83%	0.80%
2	0.99%	0.79%	0.83%
3	1.05%	0.82%	0.89%
4	1.05%	0.82%	0.89%
5	0.85%	0.84%	0.92%

Table B.20: False Positive Rates for GPU ANN

GPU ANN	10	20	30
1	0.82%	0.82%	0.78%
2	0.85%	0.82%	0.87%
3	0.84%	0.67%	0.77%
4	0.84%	0.67%	0.77%
5	0.87%	0.83%	0.87%

Table B.21: False Positive Rates for GPU SVM-High

GPU SVM-H	10	20	30
1	0.58%	0.40%	0.38%
2	0.58%	0.40%	0.38%
3	0.58%	0.40%	0.38%
4	0.58%	0.40%	0.38%
5	0.58%	0.40%	0.38%

Table B.22: False Positive Rates for CPU SVM-High

CPU SVM-H	10	20	30
1	0.83%	0.47%	0.51%
2	0.83%	0.47%	0.51%
3	0.83%	0.47%	0.51%
4	0.83%	0.47%	0.51%
5	0.83%	0.47%	0.51%

Table B.23: False Positive Rates for GPU SVM-Low

GPU SVM-L	10	20	30
1	0.62%	0.50%	0.44%
2	0.62%	0.50%	0.44%
3	0.62%	0.50%	0.44%
4	0.62%	0.50%	0.44%
5	0.62%	0.50%	0.44%

Table B.24: False Positive Rates for CPU SVM-Low

CPU SVM-L	10	20	30
1	0.52%	0.40%	0.37%
2	0.52%	0.40%	0.37%
3	0.52%	0.40%	0.37%
4	0.52%	0.40%	0.37%
5	0.52%	0.40%	0.37%

B.5 False Negative Rates

Table B.25: False Negative Rates for CPU ANN

CPU ANN	10	20	30
1	4.04%	3.92%	3.81%
2	3.97%	4.03%	3.76%
3	3.88%	3.91%	3.75%
4	3.84%	4.04%	3.88%
5	3.71%	4.00%	3.66%

Table B.26: False Negative Rates for GPU ANN

GPU ANN	10	20	30
1	4.15%	3.90%	3.87%
2	4.03%	4.02%	3.66%
3	3.94%	3.80%	3.75%
4	3.74%	4.18%	3.88%
5	3.75%	4.19%	3.63%

Table B.27: False Negative Rates for GPU SVM-High

GPU SVM-H	10	20	30
1	0.74%	0.63%	0.56%
2	0.74%	0.63%	0.56%
3	0.74%	0.63%	0.56%
4	0.74%	0.63%	0.56%
5	0.74%	0.63%	0.56%

Table B.28: False Negative Rates for CPU SVM-High

CPU SVM-H	10	20	30
1	0.72%	0.59%	0.50%
2	0.72%	0.59%	0.50%
3	0.72%	0.59%	0.50%
4	0.72%	0.59%	0.50%
5	0.72%	0.59%	0.50%

Table B.29: False Negative Rates for GPU SVM-Low

GPU SVM-L	10	20	30
1	1.37%	1.33%	0.99%
2	1.37%	1.33%	0.99%
3	1.37%	1.33%	0.99%
4	1.37%	1.33%	0.99%
5	1.37%	1.33%	0.99%

Table B.30: False Negative Rates for CPU SVM-Low

CPU SVM-L	**10**	**20**	**30**
1	2.88%	3.05%	2.29%
2	2.88%	3.05%	2.29%
3	2.88%	3.05%	2.29%
4	2.88%	3.05%	2.29%
5	2.88%	3.05%	2.29%

Appendix C: Supplemental Data for Experiment 2

C.1 R Script - Tukey Test for Experiment 2

```
#get data from files
times_40 = read.csv("Z:\\Fall\ 11\\chapter\ 4\\McPadLogs\\Combined\\40Time.csv",header=TRUE)
times_160 = read.csv("Z:\\Fall\ 11\\chapter\ 4\\McPadLogs\\Combined\\160Time.csv",header=TRUE)
#separate by dataset
times_40_0 = times_40[grep("^0$",times_40$Dataset),]
times_40_1 = times_40[grep("^1$",times_40$Dataset),]
times_40_3 = times_40[grep("3",times_40$Dataset),]
times_40_5 = times_40[grep("5",times_40$Dataset),]
times_40_7 = times_40[grep("7",times_40$Dataset),]
times_40_9 = times_40[grep("9",times_40$Dataset),]
times_40_10 = times_40[grep("^10$",times_40$Dataset),]
#separate by dataset
times_160_0 = times_160[grep("^0$",times_160$Dataset),]
times_160_1 = times_160[grep("^1$",times_160$Dataset),]
times_160_3 = times_160[grep("3",times_160$Dataset),]
times_160_5 = times_160[grep("5",times_160$Dataset),]
times_160_7 = times_160[grep("7",times_160$Dataset),]
times_160_9 = times_160[grep("9",times_160$Dataset),]
times_160_10 = times_160[grep("^10$",times_160$Dataset),]
#------------------Diff between Alg for each dataset
# diff between algs at 40 cl by dataset for training
y_train_40_0=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_0)),conf.level = 0.90)
y_train_40_0
y_train_40_1=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_1)),conf.level = 0.90)
y_train_40_1
y_train_40_3=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_3)),conf.level = 0.90)
y_train_40_3
y_train_40_5=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_5)),conf.level = 0.90)
y_train_40_5
y_train_40_7=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_7)),conf.level = 0.90)
y_train_40_7
y_train_40_9=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_9)),conf.level = 0.90)
y_train_40_9
y_train_40_10=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_40_10)),conf.level = 0.90)
y_train_40_10
# diff between algs at 40 cl by dataset for testing
y_test_40_0=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_0)),conf.level = 0.90)
y_test_40_0
y_test_40_1=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_1)),conf.level = 0.90)
y_test_40_1
y_test_40_3=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_3)),conf.level = 0.90)
y_test_40_3
y_test_40_5=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_5)),conf.level = 0.90)
y_test_40_5
y_test_40_7=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_7)),conf.level = 0.90)
y_test_40_7
y_test_40_9=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_9)),conf.level = 0.90)
y_test_40_9
```

```
y_test_40_10=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_40_10)),conf.level = 0.90)
y_test_40_10
# diff between algs at 160 cl by dataset for training
y_train_160_0=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_0)),conf.level = 0.90)
y_train_160_0
y_train_160_1=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_1)),conf.level = 0.90)
y_train_160_1
y_train_160_3=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_3)),conf.level = 0.90)
y_train_160_3
y_train_160_5=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_5)),conf.level = 0.90)
y_train_160_5
y_train_160_7=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_7)),conf.level = 0.90)
y_train_160_7
y_train_160_9=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_9)),conf.level = 0.90)
y_train_160_9
y_train_160_10=TukeyHSD(aov(lm(Train.s.~factor(Alg),times_160_10)),conf.level = 0.90)
y_train_160_10
# diff between algs at 160 cl by dataset for testing
y_test_160_0=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_0)),conf.level = 0.90)
y_test_160_0
y_test_160_1=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_1)),conf.level = 0.90)
y_test_160_1
y_test_160_3=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_3)),conf.level = 0.90)
y_test_160_3
y_test_160_5=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_5)),conf.level = 0.90)
y_test_160_5
y_test_160_7=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_7)),conf.level = 0.90)
y_test_160_7
y_test_160_9=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_9)),conf.level = 0.90)
y_test_160_9
y_test_160_10=TukeyHSD(aov(lm(Test.ms.~factor(Alg),times_160_10)),conf.level = 0.90)
y_test_160_10
#----------------diffs of dataset by ALG and CL--------------------
y_train_40_AC = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_40[grep("AC",times_40$Alg),])),conf.level = 0.90)
y_train_40_AC
y_train_40_AG = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_40[grep("AG",times_40$Alg),])),conf.level = 0.90)
y_train_40_AG
y_train_40_LH = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_40[grep("LH",times_40$Alg),])),conf.level = 0.90)
y_train_40_LH
y_train_40_GL = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_40[grep("GL",times_40$Alg),])),conf.level = 0.90)
y_train_40_GL
y_test_40_AC = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_40[grep("AC",times_40$Alg),])),conf.level = 0.90)
y_test_40_AC
y_test_40_AG = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_40[grep("AG",times_40$Alg),])),conf.level = 0.90)
y_test_40_AG
y_test_40_LH = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_40[grep("LH",times_40$Alg),])),conf.level = 0.90)
y_test_40_LH
y_test_40_GL = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_40[grep("GL",times_40$Alg),])),conf.level = 0.90)
y_test_40_GL
#diff of dataset for 160
y_train_160_AC = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_160[grep("AC",times_160$Alg),])),conf.level = 0.90)
y_train_160_AC
y_train_160_AG = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_160[grep("AG",times_160$Alg),])),conf.level = 0.90)
y_train_160_AG
```

```
y_train_160_LH = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_160[grep("LH",times_160$Alg),])),conf.level = 0.90)
y_train_160_LH
y_train_160_GL = TukeyHSD(aov(lm(Train.s.~factor(Dataset),times_160[grep("GL",times_160$Alg),])),conf.level = 0.90)
y_train_160_GL
y_test_160_AC = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_160[grep("AC",times_160$Alg),])),conf.level = 0.90)
y_test_160_AC
y_test_160_AG = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_160[grep("AG",times_160$Alg),])),conf.level = 0.90)
y_test_160_AG
y_test_160_LH = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_160[grep("LH",times_160$Alg),])),conf.level = 0.90)
y_test_160_LH
y_test_160_GL = TukeyHSD(aov(lm(Test.ms.~factor(Dataset),times_160[grep("GL",times_160$Alg),])),conf.level = 0.90)
y_test_160_GL
#-----------------Cluster comparison-----------------
#diff for AC
x_train_CL_AC_0=matrix(c(
times_40_0[grep("AC",times_40_0$Alg),]$Clusters,
times_160_0[grep("AC",times_160_0$Alg),]$Clusters,
times_40_0[grep("AC",times_40_0$Alg),]$Train.s.,
times_160_0[grep("AC",times_160_0$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_0 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_0))),conf.level = 0.90)
y_train_CL_AC_0
x_train_CL_AC_1=matrix(c(
times_40_1[grep("AC",times_40_1$Alg),]$Clusters,
times_160_1[grep("AC",times_160_1$Alg),]$Clusters,
times_40_1[grep("AC",times_40_1$Alg),]$Train.s.,
times_160_1[grep("AC",times_160_1$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_1 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_1))),conf.level = 0.90)
y_train_CL_AC_1
x_train_CL_AC_3=matrix(c(
times_40_3[grep("AC",times_40_3$Alg),]$Clusters,
times_160_3[grep("AC",times_160_3$Alg),]$Clusters,
times_40_3[grep("AC",times_40_3$Alg),]$Train.s.,
times_160_3[grep("AC",times_160_3$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_3 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_3))),conf.level = 0.90)
y_train_CL_AC_3
x_train_CL_AC_5=matrix(c(
times_40_5[grep("AC",times_40_5$Alg),]$Clusters,
times_160_5[grep("AC",times_160_5$Alg),]$Clusters,
times_40_5[grep("AC",times_40_5$Alg),]$Train.s.,
times_160_5[grep("AC",times_160_5$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_5 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_5))),conf.level = 0.90)
y_train_CL_AC_5
x_train_CL_AC_7=matrix(c(
times_40_7[grep("AC",times_40_7$Alg),]$Clusters,
times_160_7[grep("AC",times_160_7$Alg),]$Clusters,
times_40_7[grep("AC",times_40_7$Alg),]$Train.s.,
times_160_7[grep("AC",times_160_7$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_7 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_7))),conf.level = 0.90)
y_train_CL_AC_7
```

```
x_train_CL_AC_9=matrix(c(
times_40_9[grep("AC",times_40_9$Alg),]$Clusters,
times_160_9[grep("AC",times_160_9$Alg),]$Clusters,
times_40_9[grep("AC",times_40_9$Alg),]$Train.s.,
times_160_9[grep("AC",times_160_9$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_9 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_9))),conf.level = 0.90)
y_train_CL_AC_9
x_train_CL_AC_10=matrix(c(
times_40_10[grep("AC",times_40_10$Alg),]$Clusters,
times_160_10[grep("AC",times_160_10$Alg),]$Clusters,
times_40_10[grep("AC",times_40_10$Alg),]$Train.s.,
times_160_10[grep("AC",times_160_10$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AC_10 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AC_10))),conf.level = 0.90)
y_train_CL_AC_10
#diff in train by cl for AG
x_train_CL_AG_0=matrix(c(
times_40_0[grep("AG",times_40_0$Alg),]$Clusters,
times_160_0[grep("AG",times_160_0$Alg),]$Clusters,
times_40_0[grep("AG",times_40_0$Alg),]$Train.s.,
times_160_0[grep("AG",times_160_0$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_0 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_0))),conf.level = 0.90)
y_train_CL_AG_0
x_train_CL_AG_1=matrix(c(
times_40_1[grep("AG",times_40_1$Alg),]$Clusters,
times_160_1[grep("AG",times_160_1$Alg),]$Clusters,
times_40_1[grep("AG",times_40_1$Alg),]$Train.s.,
times_160_1[grep("AG",times_160_1$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_1 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_1))),conf.level = 0.90)
y_train_CL_AG_1
x_train_CL_AG_3=matrix(c(
times_40_3[grep("AG",times_40_3$Alg),]$Clusters,
times_160_3[grep("AG",times_160_3$Alg),]$Clusters,
times_40_3[grep("AG",times_40_3$Alg),]$Train.s.,
times_160_3[grep("AG",times_160_3$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_3 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_3))),conf.level = 0.90)
y_train_CL_AG_3
x_train_CL_AG_5=matrix(c(
times_40_5[grep("AG",times_40_5$Alg),]$Clusters,
times_160_5[grep("AG",times_160_5$Alg),]$Clusters,
times_40_5[grep("AG",times_40_5$Alg),]$Train.s.,
times_160_5[grep("AG",times_160_5$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_5 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_5))),conf.level = 0.90)
y_train_CL_AG_5
x_train_CL_AG_7=matrix(c(
times_40_7[grep("AG",times_40_7$Alg),]$Clusters,
times_160_7[grep("AG",times_160_7$Alg),]$Clusters,
times_40_7[grep("AG",times_40_7$Alg),]$Train.s.,
times_160_7[grep("AG",times_160_7$Alg),]$Train.s.),
```

```
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_7 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_7))),conf.level = 0.90)
y_train_CL_AG_7
x_train_CL_AG_9=matrix(c(
times_40_9[grep("AG",times_40_9$Alg),]$Clusters,
times_160_9[grep("AG",times_160_9$Alg),]$Clusters,
times_40_9[grep("AG",times_40_9$Alg),]$Train.s.,
times_160_9[grep("AG",times_160_9$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_9 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_9))),conf.level = 0.90)
y_train_CL_AG_9
x_train_CL_AG_10=matrix(c(
times_40_10[grep("AG",times_40_10$Alg),]$Clusters,
times_160_10[grep("AG",times_160_10$Alg),]$Clusters,
times_40_10[grep("AG",times_40_10$Alg),]$Train.s.,
times_160_10[grep("AG",times_160_10$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_AG_10 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_AG_10))),conf.level = 0.90)
y_train_CL_AG_10
#-------diff in LH-------
x_train_CL_LH_0=matrix(c(
times_40_0[grep("LH",times_40_0$Alg),]$Clusters,
times_160_0[grep("LH",times_160_0$Alg),]$Clusters,
times_40_0[grep("LH",times_40_0$Alg),]$Train.s.,
times_160_0[grep("LH",times_160_0$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_0 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_0))),conf.level = 0.90)
y_train_CL_LH_0
x_train_CL_LH_1=matrix(c(
times_40_1[grep("LH",times_40_1$Alg),]$Clusters,
times_160_1[grep("LH",times_160_1$Alg),]$Clusters,
times_40_1[grep("LH",times_40_1$Alg),]$Train.s.,
times_160_1[grep("LH",times_160_1$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_1 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_1))),conf.level = 0.90)
y_train_CL_LH_1
x_train_CL_LH_3=matrix(c(
times_40_3[grep("LH",times_40_3$Alg),]$Clusters,
times_160_3[grep("LH",times_160_3$Alg),]$Clusters,
times_40_3[grep("LH",times_40_3$Alg),]$Train.s.,
times_160_3[grep("LH",times_160_3$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_3 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_3))),conf.level = 0.90)
y_train_CL_LH_3
x_train_CL_LH_5=matrix(c(
times_40_5[grep("LH",times_40_5$Alg),]$Clusters,
times_160_5[grep("LH",times_160_5$Alg),]$Clusters,
times_40_5[grep("LH",times_40_5$Alg),]$Train.s.,
times_160_5[grep("LH",times_160_5$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_5 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_5))),conf.level = 0.90)
y_train_CL_LH_5
x_train_CL_LH_7=matrix(c(
times_40_7[grep("LH",times_40_7$Alg),]$Clusters,
```

```
times_160_7[grep("LH",times_160_7$Alg),]$Clusters,
times_40_7[grep("LH",times_40_7$Alg),]$Train.s.,
times_160_7[grep("LH",times_160_7$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_7 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_7))),conf.level = 0.90)
y_train_CL_LH_7
x_train_CL_LH_9=matrix(c(
times_40_9[grep("LH",times_40_9$Alg),]$Clusters,
times_160_9[grep("LH",times_160_9$Alg),]$Clusters,
times_40_9[grep("LH",times_40_9$Alg),]$Train.s.,
times_160_9[grep("LH",times_160_9$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_9 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_9))),conf.level = 0.90)
y_train_CL_LH_9
x_train_CL_LH_10=matrix(c(
times_40_10[grep("LH",times_40_10$Alg),]$Clusters,
times_160_10[grep("LH",times_160_10$Alg),]$Clusters,
times_40_10[grep("LH",times_40_10$Alg),]$Train.s.,
times_160_10[grep("LH",times_160_10$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_LH_10 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_LH_10))),conf.level = 0.90)
y_train_CL_LH_10
#-----diff of GL--=---
x_train_CL_GL_0=matrix(c(
times_40_0[grep("GL",times_40_0$Alg),]$Clusters,
times_160_0[grep("GL",times_160_0$Alg),]$Clusters,
times_40_0[grep("GL",times_40_0$Alg),]$Train.s.,
times_160_0[grep("GL",times_160_0$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_0 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_0))),conf.level = 0.90)
y_train_CL_GL_0
x_train_CL_GL_1=matrix(c(
times_40_1[grep("GL",times_40_1$Alg),]$Clusters,
times_160_1[grep("GL",times_160_1$Alg),]$Clusters,
times_40_1[grep("GL",times_40_1$Alg),]$Train.s.,
times_160_1[grep("GL",times_160_1$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_1 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_1))),conf.level = 0.90)
y_train_CL_GL_1
x_train_CL_GL_3=matrix(c(
times_40_3[grep("GL",times_40_3$Alg),]$Clusters,
times_160_3[grep("GL",times_160_3$Alg),]$Clusters,
times_40_3[grep("GL",times_40_3$Alg),]$Train.s.,
times_160_3[grep("GL",times_160_3$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_3 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_3))),conf.level = 0.90)
y_train_CL_GL_3
x_train_CL_GL_5=matrix(c(
times_40_5[grep("GL",times_40_5$Alg),]$Clusters,
times_160_5[grep("GL",times_160_5$Alg),]$Clusters,
times_40_5[grep("GL",times_40_5$Alg),]$Train.s.,
times_160_5[grep("GL",times_160_5$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_5 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_5))),conf.level = 0.90)
```

```
y_train_CL_GL_5
x_train_CL_GL_7=matrix(c(
times_40_7[grep("GL",times_40_7$Alg),]$Clusters,
times_160_7[grep("GL",times_160_7$Alg),]$Clusters,
times_40_7[grep("GL",times_40_7$Alg),]$Train.s.,
times_160_7[grep("GL",times_160_7$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_7 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_7))),conf.level = 0.90)
y_train_CL_GL_7
x_train_CL_GL_9=matrix(c(
times_40_9[grep("GL",times_40_9$Alg),]$Clusters,
times_160_9[grep("GL",times_160_9$Alg),]$Clusters,
times_40_9[grep("GL",times_40_9$Alg),]$Train.s.,
times_160_9[grep("GL",times_160_9$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_9 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_9))),conf.level = 0.90)
y_train_CL_GL_9
x_train_CL_GL_10=matrix(c(
times_40_10[grep("GL",times_40_10$Alg),]$Clusters,
times_160_10[grep("GL",times_160_10$Alg),]$Clusters,
times_40_10[grep("GL",times_40_10$Alg),]$Train.s.,
times_160_10[grep("GL",times_160_10$Alg),]$Train.s.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Train.s.")))
y_train_CL_GL_10 = TukeyHSD(aov(lm(Train.s.~factor(Clusters),as.data.frame(x_train_CL_GL_10))),conf.level = 0.90)
y_train_CL_GL_10
#--------------------------------Test times diff due to CL by ALG and Dataset
x_test_CL_AC_0=matrix(c(
times_40_0[grep("AC",times_40_0$Alg),]$Clusters,
times_160_0[grep("AC",times_160_0$Alg),]$Clusters,
times_40_0[grep("AC",times_40_0$Alg),]$Test.ms.,
times_160_0[grep("AC",times_160_0$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_0 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_0))),conf.level = 0.90)
y_test_CL_AC_0
x_test_CL_AC_1=matrix(c(
times_40_1[grep("AC",times_40_1$Alg),]$Clusters,
times_160_1[grep("AC",times_160_1$Alg),]$Clusters,
times_40_1[grep("AC",times_40_1$Alg),]$Test.ms.,
times_160_1[grep("AC",times_160_1$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_1 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_1))),conf.level = 0.90)
y_test_CL_AC_1
x_test_CL_AC_3=matrix(c(
times_40_3[grep("AC",times_40_3$Alg),]$Clusters,
times_160_3[grep("AC",times_160_3$Alg),]$Clusters,
times_40_3[grep("AC",times_40_3$Alg),]$Test.ms.,
times_160_3[grep("AC",times_160_3$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_3 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_3))),conf.level = 0.90)
y_test_CL_AC_3
x_test_CL_AC_5=matrix(c(
times_40_5[grep("AC",times_40_5$Alg),]$Clusters,
times_160_5[grep("AC",times_160_5$Alg),]$Clusters,
times_40_5[grep("AC",times_40_5$Alg),]$Test.ms.,
```

```
times_160_5[grep("AC",times_160_5$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_5 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_5))),conf.level = 0.90)
y_test_CL_AC_5
x_test_CL_AC_7=matrix(c(
times_40_7[grep("AC",times_40_7$Alg),]$Clusters,
times_160_7[grep("AC",times_160_7$Alg),]$Clusters,
times_40_7[grep("AC",times_40_7$Alg),]$Test.ms.,
times_160_7[grep("AC",times_160_7$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_7 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_7))),conf.level = 0.90)
y_test_CL_AC_7
x_test_CL_AC_9=matrix(c(
times_40_9[grep("AC",times_40_9$Alg),]$Clusters,
times_160_9[grep("AC",times_160_9$Alg),]$Clusters,
times_40_9[grep("AC",times_40_9$Alg),]$Test.ms.,
times_160_9[grep("AC",times_160_9$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_9 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_9))),conf.level = 0.90)
y_test_CL_AC_9
x_test_CL_AC_10=matrix(c(
times_40_10[grep("AC",times_40_10$Alg),]$Clusters,
times_160_10[grep("AC",times_160_10$Alg),]$Clusters,
times_40_10[grep("AC",times_40_10$Alg),]$Test.ms.,
times_160_10[grep("AC",times_160_10$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AC_10 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AC_10))),conf.level = 0.90)
y_test_CL_AC_10
#diff in test by cl for AG
x_test_CL_AG_0=matrix(c(
times_40_0[grep("AG",times_40_0$Alg),]$Clusters,
times_160_0[grep("AG",times_160_0$Alg),]$Clusters,
times_40_0[grep("AG",times_40_0$Alg),]$Test.ms.,
times_160_0[grep("AG",times_160_0$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_0 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_0))),conf.level = 0.90)
y_test_CL_AG_0
x_test_CL_AG_1=matrix(c(
times_40_1[grep("AG",times_40_1$Alg),]$Clusters,
times_160_1[grep("AG",times_160_1$Alg),]$Clusters,
times_40_1[grep("AG",times_40_1$Alg),]$Test.ms.,
times_160_1[grep("AG",times_160_1$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_1 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_1))),conf.level = 0.90)
y_test_CL_AG_1
x_test_CL_AG_3=matrix(c(
times_40_3[grep("AG",times_40_3$Alg),]$Clusters,
times_160_3[grep("AG",times_160_3$Alg),]$Clusters,
times_40_3[grep("AG",times_40_3$Alg),]$Test.ms.,
times_160_3[grep("AG",times_160_3$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_3 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_3))),conf.level = 0.90)
y_test_CL_AG_3
x_test_CL_AG_5=matrix(c(
```

```
times_40_5[grep("AG",times_40_5$Alg),]$Clusters,
times_160_5[grep("AG",times_160_5$Alg),]$Clusters,
times_40_5[grep("AG",times_40_5$Alg),]$Test.ms.,
times_160_5[grep("AG",times_160_5$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_5 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_5))),conf.level = 0.90)
y_test_CL_AG_5
x_test_CL_AG_7=matrix(c(
times_40_7[grep("AG",times_40_7$Alg),]$Clusters,
times_160_7[grep("AG",times_160_7$Alg),]$Clusters,
times_40_7[grep("AG",times_40_7$Alg),]$Test.ms.,
times_160_7[grep("AG",times_160_7$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_7 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_7))),conf.level = 0.90)
y_test_CL_AG_7
x_test_CL_AG_9=matrix(c(
times_40_9[grep("AG",times_40_9$Alg),]$Clusters,
times_160_9[grep("AG",times_160_9$Alg),]$Clusters,
times_40_9[grep("AG",times_40_9$Alg),]$Test.ms.,
times_160_9[grep("AG",times_160_9$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_9 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_9))),conf.level = 0.90)
y_test_CL_AG_9
x_test_CL_AG_10=matrix(c(
times_40_10[grep("AG",times_40_10$Alg),]$Clusters,
times_160_10[grep("AG",times_160_10$Alg),]$Clusters,
times_40_10[grep("AG",times_40_10$Alg),]$Test.ms.,
times_160_10[grep("AG",times_160_10$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_AG_10 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_AG_10))),conf.level = 0.90)
y_test_CL_AG_10
#diff in LH
x_test_CL_LH_0=matrix(c(
times_40_0[grep("LH",times_40_0$Alg),]$Clusters,
times_160_0[grep("LH",times_160_0$Alg),]$Clusters,
times_40_0[grep("LH",times_40_0$Alg),]$Test.ms.,
times_160_0[grep("LH",times_160_0$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_0 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_0))),conf.level = 0.90)
y_test_CL_LH_0
x_test_CL_LH_1=matrix(c(
times_40_1[grep("LH",times_40_1$Alg),]$Clusters,
times_160_1[grep("LH",times_160_1$Alg),]$Clusters,
times_40_1[grep("LH",times_40_1$Alg),]$Test.ms.,
times_160_1[grep("LH",times_160_1$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_1 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_1))),conf.level = 0.90)
y_test_CL_LH_1
x_test_CL_LH_3=matrix(c(
times_40_3[grep("LH",times_40_3$Alg),]$Clusters,
times_160_3[grep("LH",times_160_3$Alg),]$Clusters,
times_40_3[grep("LH",times_40_3$Alg),]$Test.ms.,
times_160_3[grep("LH",times_160_3$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
```

```
y_test_CL_LH_3 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_3))),conf.level = 0.90)
y_test_CL_LH_3
x_test_CL_LH_5=matrix(c(
times_40_5[grep("LH",times_40_5$Alg),]$Clusters,
times_160_5[grep("LH",times_160_5$Alg),]$Clusters,
times_40_5[grep("LH",times_40_5$Alg),]$Test.ms.,
times_160_5[grep("LH",times_160_5$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_5 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_5))),conf.level = 0.90)
y_test_CL_LH_5
x_test_CL_LH_7=matrix(c(
times_40_7[grep("LH",times_40_7$Alg),]$Clusters,
times_160_7[grep("LH",times_160_7$Alg),]$Clusters,
times_40_7[grep("LH",times_40_7$Alg),]$Test.ms.,
times_160_7[grep("LH",times_160_7$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_7 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_7))),conf.level = 0.90)
y_test_CL_LH_7
x_test_CL_LH_9=matrix(c(
times_40_9[grep("LH",times_40_9$Alg),]$Clusters,
times_160_9[grep("LH",times_160_9$Alg),]$Clusters,
times_40_9[grep("LH",times_40_9$Alg),]$Test.ms.,
times_160_9[grep("LH",times_160_9$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_9 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_9))),conf.level = 0.90)
y_test_CL_LH_9
x_test_CL_LH_10=matrix(c(
times_40_10[grep("LH",times_40_10$Alg),]$Clusters,
times_160_10[grep("LH",times_160_10$Alg),]$Clusters,
times_40_10[grep("LH",times_40_10$Alg),]$Test.ms.,
times_160_10[grep("LH",times_160_10$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_LH_10 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_LH_10))),conf.level = 0.90)
y_test_CL_LH_10
#diff of GL
x_test_CL_GL_0=matrix(c(
times_40_0[grep("GL",times_40_0$Alg),]$Clusters,
times_160_0[grep("GL",times_160_0$Alg),]$Clusters,
times_40_0[grep("GL",times_40_0$Alg),]$Test.ms.,
times_160_0[grep("GL",times_160_0$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_0 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_0))),conf.level = 0.90)
y_test_CL_GL_0
x_test_CL_GL_1=matrix(c(
times_40_1[grep("GL",times_40_1$Alg),]$Clusters,
times_160_1[grep("GL",times_160_1$Alg),]$Clusters,
times_40_1[grep("GL",times_40_1$Alg),]$Test.ms.,
times_160_1[grep("GL",times_160_1$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_1 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_1))),conf.level = 0.90)
y_test_CL_GL_1
x_test_CL_GL_3=matrix(c(
times_40_3[grep("GL",times_40_3$Alg),]$Clusters,
times_160_3[grep("GL",times_160_3$Alg),]$Clusters,
```

```
times_40_3[grep("GL",times_40_3$Alg),]$Test.ms.,
times_160_3[grep("GL",times_160_3$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_3 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_3))),conf.level = 0.90)
y_test_CL_GL_3
x_test_CL_GL_5=matrix(c(
times_40_5[grep("GL",times_40_5$Alg),]$Clusters,
times_160_5[grep("GL",times_160_5$Alg),]$Clusters,
times_40_5[grep("GL",times_40_5$Alg),]$Test.ms.,
times_160_5[grep("GL",times_160_5$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_5 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_5))),conf.level = 0.90)
y_test_CL_GL_5
x_test_CL_GL_7=matrix(c(
times_40_7[grep("GL",times_40_7$Alg),]$Clusters,
times_160_7[grep("GL",times_160_7$Alg),]$Clusters,
times_40_7[grep("GL",times_40_7$Alg),]$Test.ms.,
times_160_7[grep("GL",times_160_7$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_7 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_7))),conf.level = 0.90)
y_test_CL_GL_7
x_test_CL_GL_9=matrix(c(
times_40_9[grep("GL",times_40_9$Alg),]$Clusters,
times_160_9[grep("GL",times_160_9$Alg),]$Clusters,
times_40_9[grep("GL",times_40_9$Alg),]$Test.ms.,
times_160_9[grep("GL",times_160_9$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_9 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_9))),conf.level = 0.90)
y_test_CL_GL_9
x_test_CL_GL_10=matrix(c(
times_40_10[grep("GL",times_40_10$Alg),]$Clusters,
times_160_10[grep("GL",times_160_10$Alg),]$Clusters,
times_40_10[grep("GL",times_40_10$Alg),]$Test.ms.,
times_160_10[grep("GL",times_160_10$Alg),]$Test.ms.),
nrow=10,dimnames=list(c(1:10),c("Clusters","Test.ms.")))
y_test_CL_GL_10 = TukeyHSD(aov(lm(Test.ms.~factor(Clusters),as.data.frame(x_test_CL_GL_10))),conf.level = 0.90)
y_test_CL_GL_10
```

C.2 R Script - Wilcoxon Test for Experiment 2

```
scores_160 = read.csv("Z:\\Fall\_11\\chapter\_4\\McPadLogs\\Combined\\160score.csv",header=TRUE)
ac_160_df= scores_160[grep("AC",scores_160$Alg),]
ag_160_df= scores_160[grep("AG",scores_160$Alg),]
l1_160_df= scores_160[grep("LH",scores_160$Alg),]
g2_160_df= scores_160[grep("GL",scores_160$Alg),]
scores_40 = read.csv("Z:\\Fall\_11\\chapter\_4\\McPadLogs\\Combined\\40score.csv",header=TRUE)
ac_40_df= scores_40[grep("AC",scores_40$Alg),]
ag_40_df= scores_40[grep("AG",scores_40$Alg),]
l1_40_df= scores_40[grep("LH",scores_40$Alg),]
g2_40_df= scores_40[grep("GL",scores_40$Alg),]
ac_40_3_v = ac_40_df[grep("3v",ac_40_df$Config),]$Score
ac_40_3_c = ac_40_df[grep("3c",ac_40_df$Config),]$Score
ac_40_5_v = ac_40_df[grep("5v",ac_40_df$Config),]$Score
ac_40_5_c = ac_40_df[grep("5c",ac_40_df$Config),]$Score
```

```
ac_40_7_v = ac_40_df[grep("7v",ac_40_df$Config),]$Score
ac_40_7_c = ac_40_df[grep("7c",ac_40_df$Config),]$Score
ag_40_3_v = ag_40_df[grep("3v",ag_40_df$Config),]$Score
ag_40_3_c = ag_40_df[grep("3c",ag_40_df$Config),]$Score
ag_40_5_v = ag_40_df[grep("5v",ag_40_df$Config),]$Score
ag_40_5_c = ag_40_df[grep("5c",ag_40_df$Config),]$Score
ag_40_7_v = ag_40_df[grep("7v",ag_40_df$Config),]$Score
ag_40_7_c = ag_40_df[grep("7c",ag_40_df$Config),]$Score
l1_40_3_v = l1_40_df[grep("3v",l1_40_df$Config),]$Score
l1_40_3_c = l1_40_df[grep("3c",l1_40_df$Config),]$Score
l1_40_5_v = l1_40_df[grep("5v",l1_40_df$Config),]$Score
l1_40_5_c = l1_40_df[grep("5c",l1_40_df$Config),]$Score
l1_40_7_v = l1_40_df[grep("7v",l1_40_df$Config),]$Score
l1_40_7_c = l1_40_df[grep("7c",l1_40_df$Config),]$Score
g2_40_3_v = g2_40_df[grep("3v",g2_40_df$Config),]$Score
g2_40_3_c = g2_40_df[grep("3c",g2_40_df$Config),]$Score
g2_40_5_v = g2_40_df[grep("5v",g2_40_df$Config),]$Score
g2_40_5_c = g2_40_df[grep("5c",g2_40_df$Config),]$Score
g2_40_7_v = g2_40_df[grep("7v",g2_40_df$Config),]$Score
g2_40_7_c = g2_40_df[grep("7c",g2_40_df$Config),]$Score
ac_160_3_v = ac_160_df[grep("3v",ac_160_df$Config),]$Score
ac_160_3_c = ac_160_df[grep("3c",ac_160_df$Config),]$Score
ac_160_5_v = ac_160_df[grep("5v",ac_160_df$Config),]$Score
ac_160_5_c = ac_160_df[grep("5c",ac_160_df$Config),]$Score
ac_160_7_v = ac_160_df[grep("7v",ac_160_df$Config),]$Score
ac_160_7_c = ac_160_df[grep("7c",ac_160_df$Config),]$Score
ag_160_3_v = ag_160_df[grep("3v",ag_160_df$Config),]$Score
ag_160_3_c = ag_160_df[grep("3c",ag_160_df$Config),]$Score
ag_160_5_v = ag_160_df[grep("5v",ag_160_df$Config),]$Score
ag_160_5_c = ag_160_df[grep("5c",ag_160_df$Config),]$Score
ag_160_7_v = ag_160_df[grep("7v",ag_160_df$Config),]$Score
ag_160_7_c = ag_160_df[grep("7c",ag_160_df$Config),]$Score
l1_160_3_v = l1_160_df[grep("3v",l1_160_df$Config),]$Score
l1_160_3_c = l1_160_df[grep("3c",l1_160_df$Config),]$Score
l1_160_5_v = l1_160_df[grep("5v",l1_160_df$Config),]$Score
l1_160_5_c = l1_160_df[grep("5c",l1_160_df$Config),]$Score
l1_160_7_v = l1_160_df[grep("7v",l1_160_df$Config),]$Score
l1_160_7_c = l1_160_df[grep("7c",l1_160_df$Config),]$Score
g2_160_3_v = g2_160_df[grep("3v",g2_160_df$Config),]$Score
g2_160_3_c = g2_160_df[grep("3c",g2_160_df$Config),]$Score
g2_160_5_v = g2_160_df[grep("5v",g2_160_df$Config),]$Score
g2_160_5_c = g2_160_df[grep("5c",g2_160_df$Config),]$Score
g2_160_7_v = g2_160_df[grep("7v",g2_160_df$Config),]$Score
g2_160_7_c = g2_160_df[grep("7c",g2_160_df$Config),]$Score
```

Appendix D: Experimental Data for Experiment 2

D.1 Training Times for 40 Cluster Datasets

Table D.1: Training Times (s) for CPU ANN 40 Cluster Datasets

AC	0	1	3	5	7	9	10
1	6.7163	6.7153	6.8451	6.7027	6.7008	6.7166	6.7056
2	6.7086	6.6982	6.8569	6.7057	6.6991	6.6948	6.6938
3	6.7274	6.7299	6.8618	6.7135	6.7194	6.7279	6.7298
4	6.7151	6.7175	6.8580	6.7034	6.7049	6.7223	6.7193
5	6.7408	6.7314	6.8580	6.7244	6.7272	6.7340	6.7410

Table D.2: Training Times (s) for GPU ANN 40 Cluster Datasets

AG	0	1	3	5	7	9	10
1	4.0518	3.4350	3.4352	3.4323	3.4360	3.4333	3.4352
2	4.0567	3.4351	3.4354	3.4321	3.4349	3.4318	3.4347
3	4.0437	3.4346	3.4375	3.4331	3.4356	3.4321	3.4349
4	4.0548	3.4342	3.4351	3.4318	3.4356	3.4322	3.4366
5	4.0449	3.4340	3.4357	3.4327	3.4344	3.4316	3.4348

Table D.3: Training Times (s) for CPU SVM 40 Cluster Datasets

LH	0	1	3	5	7	9	10
1	0.9681	0.3316	0.4872	0.6485	0.7126	0.8429	0.9794
2	0.9855	0.3415	0.5006	0.6552	0.7163	0.8552	0.9909
3	0.9493	0.3247	0.4796	0.6369	0.7054	0.8501	0.9767
4	0.9657	0.3257	0.4820	0.6381	0.7023	0.8428	0.9821
5	0.9714	0.3325	0.4869	0.6397	0.6999	0.8327	0.9664

Table D.4: Training Times (s) for GPU SVM 40 Cluster Datasets

GL	0	1	3	5	7	9	10
1	0.3185	0.1677	0.1553	0.1786	0.1726	0.1795	0.1860
2	0.3004	0.1674	0.1557	0.1797	0.1745	0.1821	0.1911
3	0.3006	0.1620	0.1511	0.1714	0.1648	0.1734	0.1826
4	0.3049	0.1625	0.1509	0.1757	0.1750	0.1763	0.1851
5	0.3067	0.1658	0.1491	0.1679	0.1705	0.1705	0.1786

D.2 Training Times for 160 Cluster Datasets

Table D.5: Training Times (s) for CPU ANN 160 Cluster Datasets

AC	0	1	3	5	7	9	10
1	14.2613	14.3266	14.1603	14.2901	14.1922	14.1889	14.1551
2	14.2842	14.3554	14.1738	14.2443	14.1494	14.1901	14.1488
3	14.3427	14.3245	14.1998	14.2856	14.1577	14.1653	14.1323
4	14.2402	14.3114	14.1468	14.2723	14.1385	14.1699	14.1275
5	14.2780	14.3019	14.1597	14.2881	14.1562	14.2081	14.2015

Table D.6: Training Times (s) for GPU ANN 160 Cluster Datasets

AG	0	1	3	5	7	9	10
1	4.1997	3.5716	3.5748	3.5745	3.5769	3.5775	3.5781
2	4.1913	3.5715	3.5802	3.5742	3.5745	3.5719	3.5752
3	4.1976	3.5711	3.5750	3.5727	3.5763	3.5735	3.5761
4	4.1987	3.5717	3.5758	3.5716	3.5771	3.5773	3.5763
5	4.1869	3.5721	3.5775	3.5723	3.5744	3.5717	3.5746

Table D.7: Training Times (s) for CPU SVM 160 Cluster Datasets

LH	0	1	3	5	7	9	10
1	2.8058	1.8875	2.2280	2.5719	2.6965	3.0143	3.8520
2	2.8582	1.9153	2.2490	2.6124	2.7420	3.1162	3.9438
3	2.8092	1.8636	2.1899	2.5353	2.6605	3.0418	3.9120
4	2.8140	1.8826	2.2258	2.5672	2.7248	3.0905	3.9483
5	2.8466	1.8784	2.2352	2.5673	2.7069	3.1119	3.9682

Table D.8: Training Times (s) for GPU SVM 160 Cluster Datasets

GL	0	1	3	5	7	9	10
1	0.3820	0.2309	0.2184	0.2351	0.2476	0.2502	0.2517
2	0.3992	0.2366	0.2171	0.2385	0.2464	0.2597	0.2599
3	0.3916	0.2367	0.2210	0.2382	0.2535	0.2554	0.2587
4	0.4108	0.2386	0.2203	0.2423	0.2468	0.2562	0.2605
5	0.3944	0.2375	0.2173	0.2439	0.2538	0.2613	0.2636

D.3 Testing Times 40 Cluster Datasets

Table D.9: Testing Times (ms) for CPU ANN 40 Cluster Datasets

AC	0	1	3	5	7	9	10
1	3.6222	3.7133	3.5682	3.649	3.6519	3.6131	3.6549
2	3.615	3.6964	3.5657	3.6341	3.6719	3.6339	3.6787
3	3.6163	3.7058	3.5801	3.6455	3.6683	3.6232	3.6551
4	3.6211	3.7037	3.5757	3.6793	3.6713	3.6357	3.6514
5	3.6432	3.7142	3.5882	3.6465	3.6618	3.6363	3.6771

Table D.10: Testing Times (ms) for GPU ANN 40 Cluster Datasets

AG	0	1	3	5	7	9	10
1	7.3652	10.3355	7.174	10.4203	6.8803	10.1253	6.8421
2	6.8431	9.7423	6.5844	9.8516	6.6393	9.6313	6.6322
3	6.867	9.76	6.6074	9.7533	6.5536	9.6494	6.5544
4	7.5779	10.1625	7.1591	10.5081	6.9628	10.2504	6.8302
5	6.6774	9.5648	6.4981	9.572	6.4223	9.4865	6.4044

Table D.11: Testing Times (ms) for CPU SVM 40 Cluster Datasets

LH	0	1	3	5	7	9	10
1	74.3383	19.5923	30.0079	41.1312	49.3119	63.3173	78.1976
2	74.6268	19.6323	30.7686	41.1012	49.575	64.2538	78.7932
3	72.7006	19.3267	29.654	40.4885	48.8046	62.6071	77.4122
4	73.0921	19.4248	29.8266	40.7649	48.7425	62.9012	77.2125
5	73.1925	19.2266	29.6641	40.8169	49.0696	62.6285	77.6287

Table D.12: Testing Times (ms) for GPU ANN 40 Cluster Datasets

GL	0	1	3	5	7	9	10
1	57.2168	45.8018	45.0328	46.3492	45.6331	46.7354	45.7422
2	58.0419	46.8499	45.1212	47.1573	45.5681	47.9366	47.0187
3	56.2219	44.774	42.7564	45.0981	44.5871	46.1161	45.0117
4	57.1508	44.5417	43.4243	45.3192	44.4532	46.8363	45.9117
5	55.8963	44.3427	42.7401	44.6885	43.9131	46.8026	44.8443

D.4 Testing Times 160 Cluster Datasets

Table D.13: Testing Times (ms) for CPU ANN 160 Cluster Datasets

AC	0	1	3	5	7	9	10
1	10.3751	10.4083	10.4389	10.4222	10.4499	10.4243	10.4829
2	10.4585	10.44	10.4275	10.4287	10.45	10.4241	10.4768
3	10.4037	10.474	10.4653	10.4494	10.4219	10.4126	10.4034
4	10.4327	10.4541	10.425	10.3883	10.4217	10.4122	10.4257
5	10.4069	10.4233	10.4481	10.4892	10.4571	10.4636	10.4462

Table D.14: Testing Times (ms) for GPU ANN 160 Cluster Datasets

AG	0	1	3	5	7	9	10
1	20.1624	23.4153	18.9178	22.3354	19.0831	22.2119	18.9277
2	18.9125	22.0817	18.5997	21.7825	18.6926	22.1112	18.7618
3	18.8996	22.2028	18.771	22.2725	19.5977	23.4529	20.1432
4	18.9831	22.2289	18.9523	21.8945	18.7145	22.0227	18.7324
5	19.1802	22.0899	18.7078	22.1158	18.8181	22.396	19.0355

Table D.15: Testing Times (ms) for CPU SVM 160 Cluster Datasets

LH	0	1	3	5	7	9	10
1	176.4792	112.289	137.809	156.236	167.009	197.729	252.66
2	175.7645	110.811	137.993	155.227	166.283	197.175	252.215
3	175.4326	110.649	136.995	154.588	165.801	196.886	251.85
4	176.3051	111.311	137.643	155.375	166.353	197.135	252.468
5	176.0898	111.098	137.237	155.041	166.566	197.029	252.08

Table D.16: Testing Times (ms) for GPU SVM 160 Cluster Datasets

GL	0	1	3	5	7	9	10
1	69.7474	65.9597	64.6537	64.8957	65.9107	65.1321	66.9211
2	71.8845	69.9446	68.9132	69.5085	68.1857	67.8321	68.9126
3	71.0208	67.5132	66.8477	66.477	66.6972	66.4321	68.419
4	70.9263	67.2452	66.2279	66.3471	66.8389	67.2069	69.9477
5	71.8235	67.41	66.9479	67.5541	67.0898	67.5547	69.6389

D.5 Ensemble Accuracies for 40 cluster Datasets

Table D.17: Accuracies for CPU ANN Ensembles 40 for Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.5263	99.7121	99.5170	99.8050	99.3963	99.7957
2	99.5217	99.7075	99.5078	99.7539	99.3963	99.7771
3	99.5124	99.6935	99.5263	99.7864	99.4056	99.7818
4	99.5078	99.7121	99.5031	99.7678	99.3917	99.7771
5	99.5170	99.7167	99.5124	99.7632	99.4149	99.7725

Table D.18: Accuracies for GPU ANN Ensembles for 40 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.5263	99.7075	99.5124	99.7400	99.3870	99.7632
2	99.5170	99.7121	99.5078	99.7864	99.4009	99.7771
3	99.5170	99.7121	99.5170	99.7771	99.3917	99.7632
4	99.5171	99.7167	99.5170	99.7771	99.3963	99.7632
5	99.5217	99.6982	99.5217	99.7818	99.3916	99.7725

Table D.19: Accuracies for CPU SVM Ensembles for 40 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.8561	99.8561	99.8514	99.8654	99.8839	99.8793
2	99.8561	99.8561	99.8514	99.8793	99.8839	99.8886
3	99.8561	99.8561	99.8514	99.8839	99.8839	99.8793
4	99.8561	99.8561	99.8514	99.8747	99.8839	99.8793
5	99.8561	99.8561	99.8514	99.8700	99.8839	99.8793

Table D.20: Accuracies for GPU SVM Ensembles for 40 Cluster Datasets

GL	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.5960	99.7167	99.6285	99.7864	99.4985	99.7725
2	99.5867	99.7075	99.6331	99.7539	99.5031	99.7771
3	99.5914	99.7167	99.6285	99.7632	99.5031	99.7725
4	99.5867	99.7121	99.6285	99.7864	99.4938	99.7911
5	99.5960	99.7167	99.6285	99.7585	99.5031	99.7911

D.6 Ensemble Accuracies for 160 cluster Datasets

Table D.21: Accuracies for CPU ANN Ensembles for 160 Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.4660	99.4892	99.4845	99.6378	99.33591	99.6517
2	99.4567	99.4799	99.5077	99.6610	99.32662	99.6378
3	99.4706	99.4938	99.5078	99.6378	99.34056	99.6517
4	99.4520	99.4706	99.4845	99.6610	99.33592	99.6703
5	99.4659	99.4706	99.4845	99.6146	99.3545	99.7028

Table D.22: Accuracies for GPU ANN Ensembles for 160 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.4660	99.5077	99.4892	99.6471	99.34521	99.6424
2	99.4613	99.4706	99.4753	99.6749	99.33592	99.675
3	99.4520	99.4845	99.4892	99.6517	99.33128	99.6285
4	99.4613	99.4892	99.5124	99.6285	99.33129	99.6471
5	99.4659	99.4613	99.4938	99.6471	99.32663	99.6378

Table D.23: Accuracies for CPU SVM Ensembles for 160 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.8282	99.8421	99.8561	99.8607	99.84213	99.8561
2	99.8282	99.8328	99.8561	99.8607	99.84213	99.8561
3	99.8282	99.8282	99.8561	99.8607	99.84213	99.8607
4	99.8282	99.8328	99.8561	99.8607	99.84213	99.8561
5	99.8282	99.8514	99.8561	99.8561	99.84213	99.8746

Table D.24: Accuracies for GPU SVM Ensembles for 160 Cluster Datasets

GL	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	99.4567	99.4799	99.4752	99.6564	99.33129	99.6982
2	99.4474	99.4706	99.4752	99.6796	99.322	99.6982
3	99.4520	99.4381	99.4752	99.6842	99.32664	99.6982
4	99.4520	99.4752	99.4752	99.7121	99.33129	99.7074
5	99.4520	99.4567	99.4752	99.7167	99.31736	99.6796

D.7 Ensemble False Positive Rates for 40 Cluster Datasets

Table D.25: False Positive Rates for CPU ANN Ensembles for 40 Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.274%	0.228%	0.286%	0.189%	0.339%	0.202%
2	0.280%	0.241%	0.287%	0.228%	0.339%	0.221%
3	0.293%	0.241%	0.274%	0.215%	0.326%	0.202%
4	0.300%	0.228%	0.293%	0.202%	0.339%	0.221%
5	0.286%	0.222%	0.293%	0.208%	0.313%	0.215%

Table D.26: False Positive Rates for GPU ANN Ensembles for 40 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.274%	0.228%	0.293%	0.209%	0.346%	0.221%
2	0.287%	0.228%	0.286%	0.208%	0.333%	0.228%
3	0.287%	0.228%	0.293%	0.215%	0.346%	0.209%
4	0.280%	0.228%	0.293%	0.202%	0.339%	0.221%
5	0.280%	0.241%	0.286%	0.209%	0.346%	0.221%

Table D.27: False Positive Rates for CPU SVM Ensembles for 40 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.072%	0.072%	0.078%	0.078%	0.085%	0.091%
2	0.072%	0.072%	0.078%	0.085%	0.085%	0.085%
3	0.072%	0.072%	0.078%	0.078%	0.085%	0.078%
4	0.072%	0.072%	0.078%	0.078%	0.085%	0.085%
5	0.072%	0.072%	0.078%	0.078%	0.085%	0.085%

Table D.28: False Positive Rates for GPU SVM Ensembles for 40 Cluster Datasets

GL	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.326%	0.267%	0.326%	0.234%	0.339%	0.254%
2	0.333%	0.280%	0.326%	0.261%	0.332%	0.254%
3	0.333%	0.267%	0.326%	0.260%	0.339%	0.260%
4	0.333%	0.274%	0.326%	0.234%	0.339%	0.254%
5	0.326%	0.267%	0.326%	0.261%	0.339%	0.260%

D.8 Ensemble False Positive Rates for 160 Cluster Datasets

Table D.29: False Positive Rates for CPU ANN Ensembles for 160 Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.260%	0.267%	0.273%	0.280%	0.332%	0.260%
2	0.260%	0.260%	0.267%	0.254%	0.332%	0.254%
3	0.260%	0.267%	0.260%	0.300%	0.319%	0.241%
4	0.280%	0.293%	0.273%	0.280%	0.332%	0.260%
5	0.267%	0.273%	0.280%	0.273%	0.300%	0.234%

Table D.30: False Positive Rates for GPU ANN Ensembles for 160 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.273%	0.254%	0.280%	0.273%	0.312%	0.267%
2	0.267%	0.280%	0.287%	0.267%	0.332%	0.228%
3	0.274%	0.273%	0.267%	0.267%	0.326%	0.280%
4	0.260%	0.274%	0.254%	0.273%	0.332%	0.254%
5	0.267%	0.287%	0.273%	0.280%	0.332%	0.260%

Table D.31: False Positive Rates for CPU SVM Ensembles for 160 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.143%	0.123%	0.104%	0.098%	0.117%	0.104%
2	0.143%	0.136%	0.104%	0.098%	0.117%	0.098%
3	0.143%	0.136%	0.104%	0.098%	0.117%	0.091%
4	0.143%	0.124%	0.104%	0.098%	0.117%	0.098%
5	0.143%	0.111%	0.104%	0.104%	0.117%	0.078%

Table D.32: False Positive Rates for GPU SVM Ensembles for 160 Cluster Datasets

GL	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.312%	0.143%	0.326%	0.319%	0.391%	0.299%
2	0.312%	0.143%	0.326%	0.319%	0.391%	0.299%
3	0.312%	0.143%	0.326%	0.345%	0.391%	0.319%
4	0.312%	0.143%	0.326%	0.312%	0.391%	0.306%
5	0.312%	0.143%	0.326%	0.293%	0.391%	0.319%

D.9 Ensemble False Negative Rates for 40 Cluster Datasets

Table D.33: False Negative Rates for CPU ANN Ensembles for 40 Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.968%	0.437%	0.972%	0.238%	1.260%	0.212%
2	0.969%	0.420%	1.001%	0.326%	1.260%	0.228%
3	0.968%	0.469%	0.971%	0.223%	1.260%	0.261%
4	0.968%	0.436%	1.002%	0.341%	1.276%	0.227%
5	0.968%	0.437%	0.972%	0.341%	1.260%	0.260%

Table D.34: False Negative Rates for GPU ANN Ensembles for 40 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.968%	0.453%	0.972%	0.414%	1.276%	0.276%
2	0.968%	0.437%	1.002%	0.237%	1.260%	0.211%
3	0.969%	0.437%	0.958%	0.267%	1.260%	0.309%
4	0.985%	0.421%	0.958%	0.280%	1.260%	0.276%
5	0.968%	0.453%	0.958%	0.283%	1.260%	0.244%

Table D.35: False Negative Rates for CPU SVM Ensembles for 40 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.324%	0.324%	0.340%	0.280%	0.193%	0.193%
2	0.324%	0.324%	0.340%	0.189%	0.193%	0.177%
3	0.324%	0.324%	0.340%	0.189%	0.193%	0.226%
4	0.324%	0.324%	0.340%	0.267%	0.193%	0.210%
5	0.324%	0.324%	0.340%	0.281%	0.193%	0.209%

Table D.36: False Negative Rates for GPU SVM Ensembles for 40 Cluster Datasets

Run	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.595%	0.323%	0.453%	0.162%	0.904%	0.161%
2	0.611%	0.323%	0.438%	0.206%	0.905%	0.145%
3	0.595%	0.323%	0.453%	0.161%	0.888%	0.144%
4	0.611%	0.323%	0.453%	0.146%	0.920%	0.097%
5	0.595%	0.323%	0.453%	0.176%	0.888%	0.080%

D.10 Ensemble False Negative Rates for 160 Cluster Datasets

Table D.37: False Negative Rates for CPU ANN Ensembles for 160 Cluster Datasets

AC	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	1.212%	1.115%	1.121%	0.592%	1.487%	0.562%
2	1.244%	1.162%	1.046%	0.515%	1.518%	0.628%
3	1.196%	1.099%	1.061%	0.499%	1.502%	0.614%
4	1.212%	1.115%	1.105%	0.453%	1.485%	0.499%
5	1.196%	1.163%	1.090%	0.679%	1.502%	0.452%

Table D.38: False Negative Rates for GPU ANN Ensembles for 160 Cluster Datasets

AG	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	1.179%	1.084%	1.075%	0.513%	1.501%	0.583%
2	1.212%	1.148%	1.105%	0.440%	1.485%	0.565%
3	1.228%	1.115%	1.105%	0.513%	1.517%	0.597%
4	1.228%	1.097%	1.061%	0.585%	1.503%	0.597%
5	1.196%	1.163%	1.076%	0.563%	1.516%	0.613%

Table D.39: False Negative Rates for CPU SVM Ensembles for 160 Cluster Datasets

LH	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	0.243%	0.243%	0.252%	0.252%	0.259%	0.243%
2	0.243%	0.243%	0.252%	0.252%	0.259%	0.259%
3	0.243%	0.259%	0.252%	0.252%	0.259%	0.259%
4	0.243%	0.275%	0.252%	0.252%	0.259%	0.259%
5	0.243%	0.243%	0.252%	0.252%	0.259%	0.242%

Table D.40: False Negative Rates for GPU SVM Ensembles for 160 Cluster Datasets

GL	3 Vote	3 Class	5 Vote	5 Class	7 Vote	7 Class
1	1.114%	0.243%	1.000%	0.365%	1.356%	0.304%
2	1.147%	0.243%	1.000%	0.308%	1.390%	0.305%
3	1.130%	0.243%	1.000%	0.219%	1.372%	0.259%
4	1.130%	0.243%	1.000%	0.205%	1.356%	0.258%
5	1.130%	0.243%	1.000%	0.234%	1.406%	0.323%

Bibliography

[ACM99] ACM. Acm kdd cup 1999, 1999. http://www.sigkdd.org/kddcup/index.php?section=1999&method=info. Last accessed: 7 Aug 2012.

[AJK10] Malak Alshawabkeh, Byunghyun Jang, and David Kaeli. Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 104–110, New York, NY, USA, 2010. ACM.

[And80] James Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co, 1980. http://csrc.nist.gov/publications/history/ande80.pdf.

[CDD09] S. Collange, Y. S. Dandass, M. Daumas, and D. Defour. Using graphics processors for parallelizing hash-based data carving. In *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*, pages 1–10, 2009.

[ChL11] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[ChL12] Chih-Chung Chang and Chih-Jen Lin. Libsvm faq, June 2012. http://www.csie.ntu.edu.tw/~cjlin/libsvm/faq.html#f408. Last accessed: 7 Aug 2012.

[Cla09] ClamAV. Clam antivirus, 2009. http://www.clamav.net/lang/en/. Last accessed: 7 Aug 2012.

[CSK11] Andrew Cotter, Nathan Srebro, and Joseph Keshet. A gpu-tailored approach for training kernelized svms. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '11, pages 805–813, New York, NY, USA, 2011. ACM.

[CYT06] Aki Chan, Daniel Yeung, Eric Tsang, and Wing Ng. Empirical study on fusion methods using ensemble of rbfnn for network intrusion detection. In Daniel Yeung, Zhi-Qiang Liu, Xi-Zhao Wang, and Hong Yan, editors, *Advances in Machine Learning and Cybernetics*, volume 3930 of *Lecture Notes in Computer Science*, pages 682–690. Springer Berlin / Heidelberg, 2006.

[Den87] D. E. Denning. An intrusion-detection model. *Software Engineering, IEEE Transactions on*, SE-13(2):222–232, 1987.

[Don11] Luca Donati. Libcudann project page, November 2011. http://sourceforge.net/p/libcudann/project/Project/. Last accessed: July 2012.

[DFD04] L. de Sa Silva, A. C. Ferrari dos Santos, J. D. S. da Silva, and A. Montes. A neural network application for attack detection in computer networks. In *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, volume 2, pages 1569–1574, 2004.

[FCL05] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *J.Mach.Learn.Res.*, 6:1889–1918, Dec 2005.

[Fec10] Bernhard Fechner. Gpu-based parallel signature scanning and hash generation. *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*, page 1, Feb 2010.

[Fle09] Tristan Fletcher. Support vector machines explained, March 2009. http://www.tristanfletcher.co.uk/SVM%20Explained.pdf. Last accessed: 7 Aug 2012.

[GDM09] P. Garca-Teodoro, J. Daz-Verdejo, G. Maci-Fernndez, and E. Vzquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers amp; Security*, 28(12):18 – 28, 2009.

[GVK07] V. A. Golovko, L. U. Vaitsekhovich, P. A. Kochurko, and U. S. Rubanau. Dimensionality reduction and attack recognition using neural network approaches. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 2734–2739, 2007.

[Ham12] Howard Hamilton. Confusion matrix - class notes for computer science 831: Knowledge discovery in databases. university of regina, 2012. http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html. Last accessed: 7 Aug 2012.

[HHV08] Torsten Hothorn, Kurt Hornik, Mark A. van de Wiel, and Achim Zeileis. Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, 28(8):1–23, 2008.

[HuW10] Asa B. Hur and Jason Weston. A user's guide to support vector machines. pages 1–18, 2010. url=http://pyml.sourceforge.net/doc/howto.pdf. Last accessed: 7 Aug 2012.

[Khr12] Khronos. Opencl - the open standard for parallel programming of heterogeneous systems, 2012. http://www.khronos.org/opencl/. Last accessed: 15 Aug 2012.

[KoM09] C. S. Kouzinopoulos and K. G. Margaritis. String matching on a multicore gpu using cuda. In *Informatics, 2009. PCI '09. 13th Panhellenic Conference on*, pages 14–18, 2009.

[Kov10] Nicholas S. Kovach. Accelerating malware detection via a graphics processing unit, 2010. http://www.dtic.mil/dtic/tr/fulltext/u2/a529467.pdf.

[Kum07] Sailesh Kumar. Survey of current network intrusion detection techniques, December 2007. http://www.cse.wustl.edu/~jain/cse571-07/ftp/ids/index.html. Last accessed: 7 Aug 2012.

[LeS00] Wenke Lee and Salvatore J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3:227–261, 2000.

[MiI04] J. Mill and A. Inoue. Support vector classifiers and network intrusion detection. In *Fuzzy Systems, 2004. Proceedings. 2004 IEEE International Conference on*, volume 1, pages 407–410, 2004.

[Mic12] Microsoft. C++ amp, 2012. http://msdn.microsoft.com/en-us/library/hh265137(v=vs.110).aspx. Last accessed: 15 Aug 2012.

[MIT12] MIT. Darpa intrusion detection evaluation, 2012. http://www.ll.mit.edu/mission/communications/CST/darpa.html. Last accessed: 7 Aug 2012.

[MuS03] S. Mukkamala and A. H. Sung. Artificial intelligent techniques for intrusion detection. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, volume 2, pages 1266–1271, 2003.

[Nis12a] Steffen Nissen. Fann datatypes - activation function enum, 2012. http://leenissen.dk/fann/html/files/fann_data-h.html# fann_activationfunc_enum. Last accessed: 7 Aug 2012.

[Nis12b] Steffen Nissen. Fann datatypes - train enum, 2012. http://leenissen.dk/fann/html/files/fann_data-h.html#fann_train_enum. Last accessed: 7 Aug 2012.

[NiJ03] Peng Ning and Sushil Jajodia. *Intrusion Detection Techniques*. The Internet Encyclopedia. John Wiley and Sons, 2003.

[NVI11a] NVIDIA. Nvidia compute ptx: Parallel thread execution isa version 2.3. March 2011. Last accessed: 7 Aug 2012.

[NVI11b] NVIDIA. Nvidia cuda c programming guide 4.0, May 2011. Last accessed: 7 Aug 2012.

[NVI12a] NVIDIA. Nvidia cuda c best practices guide 4.1, January 2012. Last accessed: 7 Aug 2012.

[NVI12b] NVIDIA. Nvidia developer zone - cublas. 2012. http://developer.nvidia.com/cuda/cublas. Last accessed: 7 Aug 2012.

[PAF09] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks*, 53(6):864–881, April 2009.

[Per09] Roberto Perdisci. Mcpad, 2009. http://roberto.perdisci.com/projects/mcpad. Last accessed: 7 Aug 2012.

[PKS10] J. Platos, P. Kromer, V. Snasel, and A. Abraham. Scaling ids construction based on non-negative matrix factorization using gpu computing. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 86–91, 2010.

[PNY12] PNY. Gtx 280 1024mb pcie 2.0, 2012. http://www3.pny.com/font-color999999GTX-280-1024MB-PCIe-20font-P2671C396.aspx# Specifications. Last accessed: 7 Aug 2012.

[PaP07] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput.Netw.*, 51(12):3448–3470, aug 2007.

[RuN09] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[SGO09] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating gpus for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 175–184, 2009.

[ScM07] Karen A. Scarfone and Peter M. Mell. Sp 800-94. guide to intrusion detection and prevention systems (idps). Technical report, National Institute of Standards & Technology, 2007.

[Sou10] SourceFire. Snort home page, 2010. http://www.snort.org/. Last accessed: 7 Aug 2012.

[TBL09a] M. Tavallaee, E. Bagheri, Wei Lu, and A. A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Computational Intelligence for*

Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on, pages 1–6, 2009.

[TBL09b] M. Tavallaee, E. Bagheri, Wei Lu, and A. A. Ghorbani. Nsl-kdd dataset, 2009. http://www.iscx.ca/NSL-KDD/. Last accessed: 7 Aug 2012.

[Tea12] R. Core Team. R: A language and environment for statistical computing. 2012. ISBN.

[VAP08] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[ZaK09] S. Zaman and F. Karray. Collaborative architecture for distributed intrusion detection system. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–7, 2009.

[ZLM01] Zheng Zhang, Jun Li, C. N. Manikopoulos, Jay Jorgenson, and Jose Ucles. Hide: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification. In *Proc. IEEE Workshop on Information Assurance and Security*, pages 85–90, 2001.

[ZZW11] Xueqin Zhang, Chen Zhao, Jiyi Wu, and Chao Song. A gpu-rsvm based intrusion detection classifier, 2011.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
13 Sept. 2012	Master's Thesis	Sept 2010-Sept. 2012

4. TITLE AND SUBTITLE
Utilizing Graphics Processing Units for Network Anomaly Detection

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Hersack, Jonathan D, CIV

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCO/ENG/12-24

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A. Approved for Public Release; Distribution Unlimited.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This research explores the benefits of using commonly-available graphics processing units (GPUs) to perform classification of network traffic using supervised machine learning algorithms. Two full factorial experiments are conducted using a NVIDIA GeForce GTX 280 graphics card. The goal of the first experiment is to create a baseline for the relative performance of the CPU and GPU implementations of artificial neural network (ANN) and support vector machine (SVM) detection methods under varying loads. The goal of the second experiment is to determine the optimal ensemble configuration for classifying processed packet payloads using the GPU anomaly detector. The GPU ANN achieves speedups of 29x over the CPU ANN. The GPU SVM detection method shows training speedups of 85x over the CPU. The GPU ensemble classification system provides accuracies of 99% when classifying network payload traffic, while achieving speedups of 2-15x over the CPU configurations.

15. SUBJECT TERMS
CUDA, intrusion detection, machine learning, anomaly detection.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins (ENG)
U	U	U	UU	149	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x7979 barry.mullins@afit.edu